MASTER EN SCIENCES INFORMATIQUES

Etude du déterminisme des programmes Prolog avec cut, par interprétation abstraite

Mbaki Luzayisu, Efrem

Award date:
1998

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX (FUNDP)
NAMUR
INSTITUT D'INFORMATIQUE
RUE GRANDGAGNAGE, 21, B-5000 NAMUR (BELGIUM)

Etude du déterminisme des programmes Prolog avec *cut*, par interprétation abstraite.

Efrem MBAKI LUZAYISU

Mémoire présenté pour l'obtention du grade
de Licencié en Informatique

Année académique 1997 - 1998

ABSTRACT

The aim of our investigation was: the determinacy analysis of Prolog programs with cut. Knowing that use of a Prolog program receives on incoming a substitution and returns on outgoing a substitution sequence, and basing ourselves on the principles of abstract interpretation which is a general methodology for developing program analyses systematically, we considered the set of program substitution sequences as an abstract domain and the set of couples $< \beta, L >$ where β represents an abstract substitution, abstracting all concrete substitutions of a considered sequence, and L a set of "numbers" which indicate possible lengths of the same sequence.

For a better outline, we organised our text in three chapters, followed by a conclusion. The first deals with the simplified abstract domain: the abstract values are not couples but simply L lengths of solution sequences. In this domain, all concrete substitutions are abstracted by the same abstract substitution. The second exploits the results of the first chapter to deal with the generic domain, i.e. the abstract domain of couples described above. And the third and last chapter assesses, as a result of an implementation in CaML of the algorithm of abstract interpretation and adequate abstract operations, an instance of the generic abstract domain defined by $ASS =<< \mu, \tau, \rho >, L >$ where μ defines the modes, τ describes the types and ρ gives information on the sharing of variables in the Prolog program.

RESUME

Notre investigation avait comme objectif: l'étude du déterminisme de programmes Prolog avec cut. Sachant que l'utilisation d'un programme Prolog reçoit en entrée une substitution et retourne en sortie une séquence de substitutions et nous basant sur les principes d'interprétation abstraite qui se définit comme une méthodologie générale pour exprimer les méthodes d'analyse de programmes, nous avons considéré l'ensemble de séquences de substitutions programmes comme domaine abstrait et l'ensemble de couples $< \beta, L >$ où β représente une substitution abstraite, abstrayant toutes substitutions concrètes d'une séquence considérée, et L, un ensemble des "nombres" qui indiquent des éventuelles longueurs de la même séquence.

Pour une meilleure ébauche, nous avons organisons notre texte en trois chapitres, suivis d'une conclusion. Le premier traite du domaine abstrait simplifié: les valeurs abstraites ne sont pas des couples mais simplement des longueurs L des séquences solutions. Dans ce domaine, toutes les substitutions concrètes sont abstraites par une même substitution abstraite. Le deuxième exploite les résultats du premier chapitre pour traiter du domaine générique c'est-à-dire le domaine abstrait des couples décrits ci-dessus. Et le troisième et dernier chapitre évalue, en fonction d'une implémentation en CaML de l'algorithme d'interprétation abstraite et des opérations abstraites adéquates, une instance du domaine abstrait générique définie par $ASS =<< \mu, \tau, \rho >, L >$ où μ définit les modes, τ décrit les types et ρ informe sur les partages de variables dans un programme Prolog.

REMERCIEMENTS

Je profite de cette page pour m'acquitter du noble devoir de remercier toute personne qui, de loin ou de près, m'a aidé à terminer ce cycle de licence en informatique.

J'exprime particulièrement ma profonde gratitude au promoteur de ce travail, le Professeur Baudouin Le Charlier, qui n'a ménagé aucun effort pour me guider malgré ses innombrables occupations. Sans sa disponibilité et ses encouragements, cette investigation n'aurait pas pu être terminée.

Je remercie également l'assistant Christophe LECLERE pour ses interventions constructives dans la réalisation de ce memoire.

Que les familles Né-NTUMBA, MASSAMBA et BANZADIO trouvent en ce mémoire l'expression de mes remerciements pour tant d'amour qu'elles ont toujours témoigné à mon égard.

Table des matières

INTRODUCTION

L'**interprétation abstraite** (voir [4]), introduite par P. et R. Cousot (dans [3]), se définit comme une méthodologie générale pour exprimer (construire) des analyses statiques de programmes, des traitements que l'on peut appliquer à un programme en dehors de son exécution proprement dite, par exemple au cours de la compilation.

L'objectif de l'interprétation abstraite est de déduire du texte d'un programme une information permettant de l'**optimiser** telle que l'existence dans une boucle d'une expression dont la valeur reste constante lors des exécutions successives du corps de la boucle ou de **vérifier** certains aspects de correction tel que le typage des expressions du programme.

Le principe de base de la méthode est d'exécuter le programme sur un domaine "non standard" ou "abstrait", au lieu de l'exécuter sur le domaine de calcul normal. Cependant, deux hypothèses doivent être satisfaites:

1. les éléments du domaine abstrait représentent des propriétés utiles du domaine standard;
2. les calculs sur le domaine abstrait doivent être réalisés de manière suffisamment efficace et convergent en un temps fini.

Pour mieux comprendre l'idée intuitive de l'interprétation abstraite, considérons un programme P écrit dans un langage impératif (Pascal, par exemple). Une exécution de P consiste en une séquence, éventuellement infinie, d'étapes élémentaires dont chacune calcule une certaine valeur en appliquant une certaine opération à un certain nombre de valeurs initialisées ou précédemment calculées. Une telle exécution, les valeurs et les opérations qu'elle utilise sont toutes dites **concrètes**.

Supposons maintenant que nous voulions analyser le programme P pour nous assurer qu'il répond aux attentes du programmeur. Clairement, une telle analyse ne peut aboutir en observant un nombre limité d'exécutions concrètes car, en général, plusieurs valeurs concrètes sont acceptables à l'entrée de P et conduisent à des exécutions concrètes très différentes les unes des autres.

Intuitivement, l'interprétation abstraite remplace toutes les exécutions concrètes possibles en une seule dite **exécution abstraite**. Cette dernière, pour pouvoir *simuler* les exécutions concrètes et déduire automatiquement des propriétés globales nécessaires du programme P, opère non pas sur des valeurs concrètes mais sur certaines de leurs propriétés que nous appelerons dans la suite **valeurs abstraites**.

L'interprétation abstraite est présentement un domaine de recherche très actif (lire [5] et [9]), spécialement pour le paradigme déclaratif (programmation fonctionnelle, programmation logique, programmation logique avec contraintes). Et cela, d'une part, à cause du fait que ces analyses sont réellement nécessaires pour rendre les langages déclaratifs compétitifs aux langages procéduraux et, d'autre part, à cause des opportunités offertes par le caractère déclaratif de ces langages.

En programmation logique, par exemple, il existe plusieurs possibilités d'optimisation de programmes. Ces optimisations sont principalement dues à la **multidirectionnalité** des programmes logiques: tout paramètre d'une procédure peut indifféremment être utilisé comme donnée ou comme résultat ou peut être partiellement instancié. Ceci est une conséquence du **caractère déclaratif** du langage: une procédure définit une relation, elle ne précise aucune manière particulière de la calculer.

Par exemple, une procédure $di(D, I)$ qui implémente la relation "D est (une expression de) la dérivée de I ou I est une primitive de D" pourra servir à tester si D est une expression de la dérivée de I, à trouver une primitive I de D, à trouver une instance D qui soit la dérivée d'un I donné, à trouver une instance D et une instance I telles que D soit la dérivée de I,...

Chacune de ces utilisations de $di(D, I)$, relativement à la question posée, peut donner aucun, un ou plusieurs résultats. Une importante application de l'interprétation abstraite est donc de détecter le nombre des résultats que pourrait donner une utilisation particulière.

Cette application constitue l'objet de notre investigation qui sera consacrée non seulement à ***l'étude du déterminisme*** de programmes Prolog pur mais aussi au traitement de ***cut*** dans un programme Prolog. Nous nous limiterons dans ce travail à concevoir un domaine générique ainsi que les opérations abstraites adéquates que nous spécialiserons pour réaliser une évaluation. Nous deman-

$P \in$ *Programmes*
$pr \in$ *Procedures*
$c \in$ *Clauses*
$h \in$ *Tetes_de_Clause*
$g \in$ *Corps_de_Clause*
$l \in$ *Litteraux*
$b \in$ *Atomes*
$p \in$ *Noms_de_Procedure*
$f \in$ *Foncteurs*
$X_i \in$ *Variables_de_Programme*

$P ::= pr \mid pr\ P$
$pr ::= c \mid c\ pr$
$c ::= h :\text{-}\ g.$
$h ::= p(X_1, \ldots, X_n)$
$g ::= <> \mid g\ ,\ l$
$l ::= p(X_{i_1}, \ldots, X_{i_n}) \mid b$
$b ::= X_i = X_j \mid X_{i_1} = f(X_{i_2}, \ldots, X_{i_n})$

Figure 0.1: Syntaxe abstraite d'un programme Prolog pur Normalisé

derons donc aux les lecteurs intéréssés par la sémantique concrète dénotationnelle de Prolog et la sémantique abstraite correspondante basée sur la théorie des points fixes de consulter [8] où les définitions et les démonstrations sont systématiquement dévéloppées.

Avant de décrire la démarche poursuivie pour atteindre l'objectif précité, nous estimons important de parler brièvement de l'étude du déterminisme, du *cut* et de Prolog pur qui sont les trois notions composant l'ossature de notre recherche.

En effet, connue sous l'expression anglaise, *Determinacy Analysis*, l'étude du déterminisme vise à détecter les programmes qui sont déterministes (réussissant au plus une fois), ceux qui sont complètement déterministes (réussissant une et une seule fois) et ceux qui ne le sont pas. Dans ce travail, il s'agira d'indiquer le nombre de solutions possibles que peut retourner une procédure Prolog pour une question posée.

Un programme écrit en Prolog pur n'utilise qu'une et une seule opération, **l'unification**, et peut-être normalisé suivant la syntaxe abstraite donnée à la figure 0.1. Selon cette syntaxe, les programmes respectent les quatre restrictions suivantes:

1. toutes les variables utilisées sont de la forme X_i, c'est-à-dire composée d'une lettre X et d'un indice entier i. Ces variables sont dites **variables de programme**;

2. une tête de clause est de la forme $p(X_1, \ldots, X_n)$;

3. si une clause a m variables, ces dernières sont $X_1, \ldots, X_m$;

4. les atomes dans le corps d'une clause sont de l'une des trois formes suivantes: $X_i = X_j$, $X_i = f(X_{i_1}, \ldots, X_{i_n})$ ou $p(X_{i_1}, \ldots, X_{i_n})$, où les variables sont deux à deux distinctes dans chaque cas.

Le *cut*, noté !, est une pseudo opération dont l'exécution dans une clause d'une procédure Prolog impose le non retour (en arrière) sur les atomes précédents et la non exécution des clauses suivantes. Il est généralement utilisé pour optimiser les programmes prolog.

Parmi tant de travaux publiés sur ce sujet (l'étude du déterminisme), nous considérons celui de D. Salhin [10] que nous exprimons comme une instance de l'interprétation abstraite de Prolog basée sur les séquences de substitutions. Nos valeurs abstraites *(Abstract Substitution Sequences (ASS))* sont des couples $< \beta, L >$ où β représente une substitution abstraite, abstrayant toutes les substitutions concrètes de la séquence concrète considérée, et L, un ensemble de "*nombres*" qui indiquent les éventuelles longueurs[1] de la séquence concrète.

Pour une meilleure ébauche, nous avons organisé la suite en trois chapitres suivis d'une conclusion. Le premier traite du domaine abstrait simplifié: les valeurs abstraites ne sont pas des couples mais simplement des longueurs L des séquences. Dans ce domaine, on suppose que toutes les séquences concrètes sont abstraites par une seule substitution abstraite. Le deuxième chapitre exploite les résultats du premier chapitre pour traiter du domaine générique, c'est-à-dire le domaine abstrait des couples décrits ci-dessus. Et le troisième et dernier chapitre évalue, en fonction d'une implémentation en CaML de l'algorithme d'interprétation abstraite ([8] et [7]) et des opérations abstraites adéquates, une instance du domaine abstrait générique définie par $ASS =<< \mu, \tau, \rho >, L >$ où μ définit les modes des termes correspondants aux variables de programme , τ décrit leurs types et ρ informe sur les interdépendances des variables dans un programme Prolog.

Le choix du CaML se justifie, d'une part, par le souci d'exhiber une application concrète de ce langage fonctionnel souvent utilisé pour des fins pédagogiques et, d'autre part, par notre attachement au caractère mathématique de l'algorithme d'interprétation abstraite qui est principalement fondé sur des opérations abstraites, et donc sur des fonctions relatives aux valeurs abstraites.

[1] La longueur d'une séquence des substitutions est le nombre de substitutions qu'elle contient.

Chapitre 1

DOMAINE ABSTRAIT SIMPLIFIE

Etant donné que l'utilisation d'un programme Prolog reçoit en entrée une substitution et donne en sortie une séquence de substitutions programmes, le principe de l'interprétation abstraite, par rapport au souci de réaliser une étude du déterminisme des programmes Prolog, nous suggère de considérer l'ensemble des séquences des substitutions programmes (PSS) comme domaine concret et comme domaine abstrait, l'ensemble des substitutions abstraites définies par des couples $< \beta, L >$ où β représente une substitution abstraite, abstrayant toutes les substitutions concrètes de la séquence programme considérée, et L, un ensemble de *nombres* qui indiquent les éventuelles longueurs de la même substitution programme.

Dans un premier temps, nous nous sommes proposés de définir les opérations abstraites en simplifiant le domaine abstrait de telle sorte qu'il y ait **une et une seule substitution abstraite pour toutes les substitutions programmes**. Dans la suite de ce chapitre, cette unique substitution abstraite est notée β.

Mais remarquons avant toute chose qu'une substitution programme est un objet de la forme:

$$\theta = \{X_{i_1}/t_{i_1}, \ldots, X_{i_n}/t_{i_n}\}$$

où les i_j sont deux à deux distincts et les t_{i_j} des termes et que

$$PSS = \{S =< \theta_1, \theta_2, \ldots, \theta_n, - >: \theta_i \ (1 \leq i \leq n) \text{ est une substitution programme}\},$$

où la notation $S =< \theta_1, \theta_2, \ldots, \theta_n, - >$ sous-entend les trois cas suivants:

1. $S =< \theta_1, \theta_2, \ldots, \theta_n >$ (S **finie**). Une telle séquence est obtenue lorsque les substitutions répondant à la question posée existent en nombre fini;

2. $S =< \theta_1, \theta_2, \ldots, \theta_n, \perp >$ (S **incomplète**). Une telle séquence est obtenue lorsque la procédure affiche un certain nombre de substitutions réponses puis boucle;

3. $S =< \theta_1, \theta_2, \ldots, \theta_n, \ldots >$ (S **infinie**). Une telle séquence est obtenue lorsqu'il existe une infinité de substitutions répondant à la question posée.

Il est aussi important de noter que deux substitutions programmes ne peuvent être composées ni être utilisées pour la recherche d'un unificateur général. Par contre, pour une substitution programme θ et une substitution standard (usuelle) σ, nous définissons:

$$\theta\sigma = \{X_{i_1}/t_{i_1}\sigma, \ldots, X_{i_n}/t_{i_n}\sigma\}.$$

Pour éviter toute confusion, on considère que l'ensemble des variables de programme de θ, ***dom(θ)***, est disjoint de l'ensemble des variables (standards) contenues dans t_{i_j}, ***codom (θ)***. Dans la suite, nous noterons les variables standards par la lettre y munie d'un indice.

Par ailleurs, si θ_1 et θ_2 sont deux substitutions programmes, nous notons $\theta_1 \leq \theta_2$ pour signifier qu'il existe une substitution standard σ vérifiant l'égalité $\theta_2 = \theta_1\sigma$.

1.1 Domaine Abstrait

1.1.1 Séquences Abstraites

L'ensemble des Séquences de Substitutions Abstraites est défini par:
$ASS = \wp(AASS)$ où $AASS$ contient des **séquences abstraites atomiques** et est défini par:

$$AASS = \{BC, 0, 1, 1^+, 2, 2^+\}^1.$$

où

- BC représente le bouclage. Il est utilisé pour signifier qu'une procédure boucle sans donner de réponses;
- 0 indique l'absence de substitutions répondant à la question posée;
- 1 signifie l'existence et l'affichage d'une et une seule solution;
- 1^+ sous-entend l'affichage d'une solution avant que la procédure boucle;
- 2 indique l'existence d'un nombre fini n $(2 \leq n \in \mathbb{N})$ de solutions;
- 2^+ signifie soit l'affichage de n $(2 \leq n \in \mathbb{N})$ solutions avant le bouclage ou soit l'existence d'une infinité de solutions.

Cela étant, nous définissons la fonction de concrétisation $Cc : AASS \longrightarrow \wp(PSS)$ de la manière suivante:

$$\begin{aligned}
Cc(BC) &= \{< \perp >\} \\
Cc(0) &= \{< >\} \\
Cc(1) &= \{S \in PSS | Ns(S) = 1 \text{ et } S \text{ est finie }\} \\
Cc(1^+) &= \{S \in PSS | Ns(S) = 1 \text{ et } S \text{ est incomplète}\} \\
Cc(2) &= \{S \in PSS | Ns(S) > 1 \text{ et } S \text{ est finie}\} \\
Cc(2^+) &= \{S \in PSS | Ns(S) > 1 \text{ et } S \text{ est infinie ou incomplète}\}.
\end{aligned}$$

Et donc, la concrétisation $Cc : ASS \longrightarrow \wp(PSS)$ est définie par:

$$Cc(B) = \bigcup_{b \in B} Cc(b) \quad \forall B \in ASS$$

[1] Dans [10], ces éléments sont respectivement notés $\{\pounds, 0, 1, 1', 2, 2'\}$.

Remarquons en passant que la fonction Cc est monotone (croissante) par rapport à la relation d'inclusion ensembliste induite dans ASS, que nous noterons dans la suite $\subseteq$.

1.1.2 Séquences Abstraites avec information sur le *cut*

En Prolog, un *cut* dans une des clauses d'une procédure donnée n'est pas forcément exécuté pour toute utilisation de la procédure. Etant donné que son exécution ou sa non exécution influence fortement le résultat final, nous munissons les séquences atomiques, et donc les séquences abstraites, d'une composante supplémentaire qui informera sur une éventuelle exécution d'un *cut*.

Une séquence atomique sera munie de la valeur abstraite *cut* pour signifier qu'elle exprime la *longueur* d'une séquence concrète qui a subit une exécution d'un *cut*. Sinon, elle sera munie de la valeur *nocut*.

Avant de définir la fonction de concrétisation correspondant aux séquences munies de l'information sur le *cut*, rappelons que l'exécution d'un *cut* présent dans une clause *tête* : $-a_1, a_2, \ldots, a_{j-1}, !, a_{j+1}, \ldots, a_p$ impose le non retour aux atomes qui le précèdent $(a_1, a_2, \ldots, a_{j-1})$ et la non exécution des clauses suivantes si la clause courante donne une solution.

Cela étant, nous définissons l'ensemble des séquences de Substitutions Abstraites avec information sur le *cut* par: $ASSC = \wp(AASS \times CF)$ où $CF = \{cut, nocut\}$.

La fonction de concrétisation $Cc : ASSC \longrightarrow \wp(PSS)$ est définie par:

$$Cc(B) = \bigcup_{<b,cf> \in C} Cc(< b, cf >) \quad \forall C \in ASSC$$

où $Cc(< b, cf >) = \{< S, cf >: S \in Cc(b)\}$.

1.2 Opérations Abstraites

1.2.1 L'élargissement

Signature: $[ASS \times ASS \longrightarrow ASS]$

Intuitivement, cette opération vise à observer l'évolution de séquences de substitutions entre deux itérations consécutives pour garantir la convergence quand une bonne précision semble être obtenue.

Dans ce domaine, une séquence de substitutions abstraites B_i produite à l'étape i serait intuitivement différente de la séquence B_{i-1} de l'étape $i-1$ par une éventuelle disparition de certains éléments "incomplets" ($BC, 1^+$ ou 2^+) de B_{i-1} qui seraient remplacés par des éléments "complets" ($0, 1$ ou 2) dans B_i.

De ce fait, il est logique que tout calcul puisse commencer par $\{BC\}$ de telle sorte que l'algorithme fonctionne jusqu'à l'élimination du maximum possible d'éléments incomplets avant de forcer sa terminaison.

Pour formaliser cette idée, nous avons besoin d'une relation de pré-ordre, $\sqsubseteq$, dans ASS que nous définissons intuitivement de la manière suivante: $B_1 \sqsubseteq B_2$ si et seulement si B_2 contient des éléments "plus complets" que certains éléments de B_1 et B_1 contient des éléments "moins complets" que certains éléments de B_2.

L'écriture mathématique de cette relation de pré-ordre de calcul utilise la relation **"est moins complet que"** notée $\sqsubseteq$ et définie dans $AASS$ de la manière suivante:

$\forall\, b_1,\, b_2, \in AASS$

$$b_1 \sqsubseteq b_2 \quad ssi \quad b_1 = b_2 \text{ ou } b_1 \sqsubset b_2$$

où $BC \sqsubset 0$, $BC \sqsubset 1^+$, $1^+ \sqsubset 1$, $1^+ \sqsubset 2^+$ et $2^+ \sqsubset 2$.

Définition [Pré-ordre de calcul, $\sqsubseteq$]

Soient $B_1,\ B_2 \in ASS$.

$$B_1 \sqsubseteq B_2 \Longleftrightarrow \left\{ \begin{array}{c} (\forall b_1 \in B_1 :\ \exists b_2 \in B_2 \text{ tel que } b_1 \sqsubseteq b_2) \\ \wedge \\ (\forall b_2 \in B_2 :\ \exists b_1 \in B_1 \textit{ tel que } b_1 \sqsubseteq b_2) \end{array} \right\}$$

Nous écrirons $B_1 \sqsubset B_2$ pour signifier que $B_1 \sqsubseteq B_2$ et $(B_2 \not\sqsubseteq B_1)$

Implémentation de l'Elargissement $[B_{new} = B \nabla B_{old}]$

$$B_{new} = \left\{ \begin{array}{ll} B & \text{si } B_{old} \sqsubset B \\ B \cup B_{old} & \text{sinon} \end{array} \right.$$

Remarquons en passant que l'opération ainsi implémentée ne correspond pas tout à fait à l'idée spécificiée ci-dessus car elle ne marche que si $B_{old} \sqsubseteq B$ à chaque fois qu'elle est appliquée. Avant de discuter sur le résultat obtenu par l'Elargissement, définissons d'abord l'équivalence de deux Séquences Abstraites et l'ordre de calcul renforcé et énonçons la condition de convergence de l'Elargissement.

Définition [Séquences Abstraites équivalentes]

Intuitivement, nous dirons que deux séquences de substitutions abstraites B_1 et B_2 sont équivalentes lorqu'elles ont un même "degré" de précision: B_1 est à la fois plus complet et moins complet que B_2.

Soient $B_1,\ B_2 \in ASS$,

$$B_1 \approx B_2 \quad ssi \quad B_1 \sqsubseteq B_2 \textit{ et } B_2 \sqsubseteq B_1$$

Remarquons que $\approx$ est une relation d'équivalence car $\sqsubseteq$ est un pré-ordre. Signalons en passant que $\approx$ admet 42 classes d'équivalence dont 28 admettent un seul élément , 10 sont à deux éléments et 4 à quatre éléments. De plus, il est important de noter que deux séquences abstraites équivalentes distinctes ont des concrétisations différentes.

Définition [ordre de calcul renforcé, $\trianglelefteq$]

Soient B_1, $B_2 \in ASS$,

$$B_1 \trianglelefteq B_2 \quad ssi \quad B_1 \sqsubset B_2 \; ou \; (B_1 \approx B_2 \; et \; B_1 \subseteq B_2)$$

Remarquons que $\trianglelefteq$ est une relation d'ordre; par conséquent toute chaîne $B_1 \trianglelefteq B_2 \ldots B_i \trianglelefteq \ldots$ est stationnaire car ASS est fini.

Proposition [Convergence de l'Elargissement]

Si $\{B_i\}_{i\in\mathbb{N}}$ et $\{B'_i\}_{i\in\mathbb{N}}$ sont deux suites d'éléments de ASS telles que:

1. $B'_i \sqsubseteq B_{i+1} \quad \forall i \in \mathbb{N}$ et
2. $B'_{i+1} = B_{i+1} \nabla B'_i \quad \forall i \in \mathbb{N}$,

alors $B_i \sqsubset B'_i \; \forall i \in \mathbb{N}$ et que la suite $\{B'_i\}_{i\in\mathbb{N}}$ est stationnaire [2].

Si toutes les opérations abstraites sont compatibles par rapport à la relation $\sqsubseteq$, chaque itération de l'algorithme d'interprétation abstraite satisferait la condition $B_o \sqsubseteq B$, où B_o est la séquence courante et B, la séquence calculée. Dans ces conditions, la proposition ci-dessus garantit la convergence de l'algorithme d'interprétation abstraite.

Il est possible de modifier légèrement notre cadre mathématique de manière à garantir la condition $B_o \sqsubseteq B$ même si certaines opérations abstraites ne sont pas compatibles avec la relation $\sqsubseteq$. La solution triviale serait de vérifier si $B_o \sqsubseteq B$ avant chaque application de l'Elargissement. Si cette condition n'est pas satisfaite, L'Elargissement consisterait à fusionner des résultats successifs.

1.2.2 L'entrée d'une clause

Signature: $[EXTC(c,.) : AS \longrightarrow ASSC]$

Déroulement concret

Cette opération est exécutée à l'entrée d'une clause. Etant donné une clause c et une substitution programme θ, elle procède en ces trois étapes suivantes:

1. Retrouver toutes les variables de c non contenues dans la tête (de c);

[2] La démonstration de cette proposition se trouve dans [8]

$nat(X_1)$:- $X_1 = 0$. *(Clause 1)*
$nat(X_1)$:- $X_1 = s(X_2)$, $nat(X_2)$. *(Clause 2)*

Figure 1.1: Programme Prolog définissant les entiers naturels

2. Associer une nouvelle variable standard à chacune des variables trouvées à l'étape 1 ;

3. Insérer dans θ les "couples" trouvés à l'étape 2.

Exemple

Pour la procédure *nat* donnée à la figure 1.1, nous pouvons calculer:
$EXTC(clause1, \{X_1/y_1\}) =< \{X_1/y_1\}, nocut >$ et
$EXTC(clause2, \{X_1/y_1\}) =< \{X_1/y_1, X_2/y_2\}, nocut >$.

Spécification

Soient $\beta \in AS$, $\theta \in PSS$

$$\theta \in Cc(\beta) \Longrightarrow EXTC(c, \theta) \in Cc(EXTC(c, \beta)).$$

Implémentation

$$EXTC(c, \beta) = \{< 1, nocut >\}$$

Correction

Soit $\theta' \in PSS$ telle que $x_i\theta' = x_i\theta$ $(\forall\ i : 1 \leq i \leq n)$ et $x_{n+1}\theta', \ldots, x_m\theta'$ des variables standards distinctes non contenues dans $codom(\theta)$. Nous avons que

$$\begin{aligned} EXTC(c, \theta) &= << \theta' >, nocut > \\ &\in Cc(\{< 1, nocut >\}) \\ &= Cc(EXTC(c, \beta))\odot \end{aligned}$$

1.2.3 La sortie d'une clause

Signature: $[RESTRC(c,.) : ASSC \longrightarrow ASSC]$

Déroulement concret

Cette opération est exécutée à la sortie d'une clause. Elle efface dans les substitutions contenues dans la séquence programme de sortie tous les "couples" dont les variables de programme ne figurent pas dans la tête de la clause.

Exemple

Pour la procédure *nat* donnée à la figure 1.1, nous avons par exemple:
$RESTRC(clause2, < \{X_1/y_1, X_2/y_2\}, nocut >) =< \{X_1/y_1\}, nocut >$

Spécification

Soient $C \in ASSC$ et $< S, cf > \in (PSS \times CF)$,

$< S, cf > \in Cc(C) \Longrightarrow RESTRC(c, < S, cf >) \in Cc(RESTRC(c, C))$.

Implémentation

$$RESTRC(c, C) = \bigcup_{<b,cf> \in C} RESTRC_{suppl}(< b, cf >)$$

où $RESTRC_{suppl}(< b, cf >) =< b, cf >$

Correction de $RESTRC_{aux}$

Lorsque $b \in \{BC, 0, 1, 1^+, 2^+\}$, il est évident que toute restriction d'une séquence programme sur l'ensemble des variables de la tête d'une clause ne modifie ni le nombre des substitutions programmes contenues dans la séquence ni l'information sur le *cut*. Par contre si $b = 2$, on peut craindre que lorsque la séquence considérée contient exactement deux substitutions, les restrictions de ces dernières sur l'ensemble des variables de la tête de la clause soient égales. Mais, même dans ce cas, le nombre des substitutions reste intact car une séquence est différente d'un ensemble ($Longueur(< \theta, \theta >) = 2$ et $\#\{\theta, \theta\} = 1) \odot$

1.2.4 La préparation à un appel

Signature: $[RESTRG(l,.) : AS \longrightarrow AS]$

Déroulement concret

Cette opération prépare une substitution programme aux conditions d'appel. Si $p(X_{i_1}, \ldots, X_{i_k})$ est l'atome appelant et $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ est la substitution programme courante, $RESTRG$ retournera $\{X_{i_1}/t_{i_1}, \ldots, X_{i_k}/t_{i_k}\}$ où les t_{i_j} sont des termes correspondant aux X_{i_j} dans θ.

Exemple

Pour la procédure *nat* donnée à la figure 1.1, nous pouvons calculer:
$RESTRG(nat(X_2), \{X_1/valeur_1, X_2/valeur_2\}) = \{X_1/valeur_2\}$.
Remarquons que $RESTRG$ n'est pas définie si le nombre de variables présentes dans le littéral d'appel est plus grand que le cardinal du *dom* de la substitution considérée.

Spécification

Soient $\beta \in AS$ *et* $\theta \in PSS$,

$\theta \in Cc(\beta) \Longrightarrow RESTRG(l,\theta) \in Cc(RESTRG(l,\beta))$.

Implémentation

$RESTRG(l,\beta) = \beta$

Correction

C'est évident car notre AS est un singleton; RESTRG(l,β) (et θ) appartien(nen)t à la concrétisation de β, l'unique élément de AS $\odot$

1.2.5 L'unification de deux variables

Signature: $[UNIF\_VAR(X_i = X_j, .) : AS \longrightarrow ASS]$

Déroulement concret

Si X_i et X_j sont deux variables de programme présentes dans une substitution programme θ, l'opération UNIF_VAR unifie les termes t_i et t_j et retourne $< \theta\sigma >$ si σ est la substitution standard résultant de l'unification et $< >$ si les deux termes ne sont pas unifiables.

Exemple

En prenant $\theta = \{X_1/y_1, X_2/s(y_1), X_3/valeur\}$, nous calculons:

$$\begin{aligned} UNIF\_VAR(X_1 = X_3, \theta) &= < \{X_1/y_1, X_2/s(y_1), X_3/valeur\}\{y_1/valeur\} > \\ &= < \{X_1/y_1\{y_1/valeur\}, X_2/f(y_1)\{y_1/valeur\}, \\ &\quad X_3/valeur\{y_1/valeur\}\} > \\ &= < \{X_1/valeur, X_2/s(valeur), X_3/valeur\} > . \end{aligned}$$

Spécification

Soient $\beta \in AS$ *et* $\theta \in PSS$,

$\theta \in Cc(\beta) \Longrightarrow UNIF\_VAR(X_i = X_j, \theta) \in Cc(UNIF\_VAR(X_i = X_j, \beta)$.

Implémentation

$UNIF\_VAR(X_i = X_j, \beta) = \{0,\ 1\}$

Correction

C'est trivial car $Cc(\{0,1\})$ contient les deux résultats possibles de $UNIF\_VAR\odot$

1.2.6 L'unification d'une variable et d'un foncteur

Signature: $[UNIF\_FUNC(X_1 = f(X_2, \ldots, X_n), .) : AS \longrightarrow ASS]$

Déroulement concret

UNIF_FUNC($X_1 = f(X_2, \ldots, X_n)$ retourne $< >$ si les deux termes ne sont pas unifiables et $< \theta\sigma >$ si σ est l'unificateur général de nos deux termes.

Exemples
En prenant $\theta = \{X_1/y_1, X_2/s(y_1), X_3/0\, X_4/s(0)\}$, nous calculons:

$$\begin{aligned} UNIF\_VAR(X_2 = s(X_3), \theta) &= < \{X_1/y_1, X_2/s(y_1), X_3/0, X_4/s(0)\}\{y_1/0\} > \\ &= < \{X_1/y_1\{y_1/0\}, X_2/f(y_1)\{y_1/0\}, X_3/0\{y_1/0\}, \\ &\quad X_4/s(0)\{y_1/0\}\} > \\ &= < \{X_1/0, X_2/s(0), X_3/0, X_4/s(0)\} > \text{ et} \end{aligned}$$

$$\begin{aligned} UNIF\_VAR(X_3 = s(X_4), \theta) &= < \{X_1/y_1, X_2/s(y_1), X_3/0, X_4/s(0)\}\{\ \} > \\ &= < > . \end{aligned}$$

$p(X_1, X_2)$:- 1 q(X_1), 2 q(X_2), 3 ! 4 . 5
$p(X_1, X_2)$:- 6 $X_1 = X_2$ 7 .8
$q(X_1)$:- $X_1 = a$.
$q(X_1)$:- $X_1 = b$.

Figure 1.2: Un programme normalisé contenant un cut et muni des points de programme.

Première clause de p	
$S_1 =< \{X_1/y_1, X_2/y_2\} >$	$cf_1 = nocut$
$S_2 =< \{X_1/a, X_2/y_2\}, \{X_1/b, X_2/y_2\} >$	$cf_2 = nocut$
$S_3 =< \{X_1/a, X_2/a\}, \{X_1/a, X_2/b\}, \{X_1/b, X_2/a\}, \{X_1/b, X_2/a\} >$	$cf_2 = nocut$
$S_4 =< \{X_1/a, X_2/a\} >$	$cf_4 = cut$
Deuxième clause de p	
$S_5 =< \{X_1/a, X_2/a\} >$	$cf_5 = cut$
$S_6 =< \{X_1/y_1, X_2/y_2\} >$	$cf_6 = nocut$
$S_7 =< \{X_1/y_1, X_2/y_1\} >$	$cf_7 = nocut$
$S_8 =< \{X_1/y_1, X_2/y_1\} >$	$cf_8 = nocut$

Figure 1.3: Les séquences en différents points de programmes.

Spécification

Soient $\beta \in AS$ *et* $\theta \in PSS$.
$\theta \in Cc(\beta) \Longrightarrow UNIF\_FUNC(X_1 = f(X_2, \ldots, X_n), \theta) \in Cc(UNIF\_FUNC(\beta))$.

Implémentation

$UNIF\_FUNC(X_1 = f(X_2, \ldots, X_n), \beta) = \{0,\ 1\}$

Correction

C'est trivial car Cc({0,1}) contient les deux résultats possibles de UNIF_FUNC⊙

1.2.7 L'interprétation abstraite de cut

Signature: $[AI\_CUT : ASSC \longrightarrow ASSC]$

Déroulement concret

Rappelons une fois de plus que l'exécution d'un *cut* consiste en une imposition de non retour (en arrière) sur les atomes précédents et en la non exécution des clauses suivantes.

Exemple

Les figures 1.2 et 1.3 montrent que S_4 est obtenue de S_3 après application du *cut*. En réalité, un compilateur Prolog ne complète pas les séquences intermédiaires suivant la logique du tableau 1.3. Pour la première clause de p, un compilateur Prolog donnerait au point 2 la substitution $\{X_1/a, X_2/y_2\}$ après exécution de l'atome $q(X_1)$. Ensuite, il utiliserait ce résultat comme entrée du prochain appel de q et trouverait $\{X_1/a, X_2/a\}$ au point 3.

En l'absence du *cut*, il serait retourner au point 2 afin d'explorer le prochain résultat du $q(X_2)$ avec le même terme pour X_1. Il ferait ce va-et-vient jusqu'à l'épuissement de toutes les solutions de $q(X_2)$. Puis, il retournerait au point 1 pour prendre la deuxième solution de $q(X_1)$ et refaire le même parcourt. Il continuerait ainsi jusqu'à ce que toutes les valeurs possibles de X_1 soient exploitées.

Mais le *cut* l'obligera à s'arrêter et à afficher la séquence courante. Toutefois, si cette dernière était vide, il continuerait sa racherche en exécutant la seconde clause de p.

Par ailleurs, si une coïncidence faisait qu'un atome précédant un *cut* boucle, le *cut* ne sera jamais exécuté et la procédure toute entière bouclerait.

Spécification

Soient $C \in ASSC\ \theta \in PSS\ et\ cf \in CF$,

$$\begin{aligned} <> \in Cc(C) &\implies << >, cf > \in Cc(AI\_CUT(C)); \\ < \bot, cf > \in Cc(C) &\implies << \bot >, cf > \in Cc(AI\_CUT(C)) ; \\ < \theta :: S, cf > \in Cc(C) &\implies << \theta >, cut > \in Cc(AI\_CUT(C)). \end{aligned}$$

Implémentation

$\forall C \in ASSC$,

$$AI\_CUT(C) = \bigcup_{<b,cf>\in C} AI\_CUT_{aux}(< b, cf >)$$

où

$$AI\_CUT_{aux}(< b, cf >) = \begin{cases} < b, cf > & \text{si } b \in \{BC, 0\} \\ < 1, cut > & \text{si } b \in \{1, 1^+, 2, 2^+\} \end{cases}$$

Correction

Les deux premiers cas sont triviaux. Supposons maintenant que $S =< \theta :: S', cf > \in Cc(C)$. Par la définition de la concrétisation, nous savons qu'il existe $< b, cf >$ dans C avec $b \in \{1, 1^+, 2, 2^+\}$ tel que $S \in Cc(< b, cf >)$. Il s'en suit que $< \theta, cut > \in Cc(< 1, cut >) = Cc(AI\_CUT(< b, cf >))$. Et donc $< \theta, cut > \in Cc(AI\_CUT(C))$ car ,de ce qui précède, $< b, cf > \in C\odot$

1.2.8 L'extraction de la séquence

Signature: $[SEQ : ASSC \longrightarrow ASS]$

Déroulement concret

Pour un couple $< S, cf > \in (PSS \times CF)$ reçu en argument, SEQ retourne S. Elle transforme une séquence programme munie de l' information sur le *cut* en une séquence programme simple en ignorant simplement l'information sur le *cut* .

Spécification

Soient $C \in ASSC, S \in PSS$ et $cf \in CF$,

$< S, cf > \in Cc(C) \Longrightarrow S \in Cc(SEQ(C))$.

Implémentation

Soit $< b, cf > \in \ (AASS \times CF)$. Nous définissons l'opération auxiliaire SEQ_{aux} par:

$$SEQ_{aux}(< b, cf >) = b$$

Ainsi, $\forall C \in ASSC$, on a:

$$SEQ(C) = \bigcup_{<b,cf> \in C} SEQ_{aux}(< b, cf >) = \{b : \exists < b, cf > \in C\}$$

Correction

Soit $< S, cf > \in \ Cc(C)$. Par la définition de la concrétisation, il existe $< b, cf >$ dans C tel que $< S, cf > \in \ Cc(< b, cf >)$. Par suite, $S \in Cc(b) = Cc(SEQ(< b, cf >))$. Et comme $Cc(SEQ(< b, cf >)) \subseteq Cc(SEQ(C))$ par la définition de Cc, nous concluons que $S \in Cc(SEQ(C))\odot$

1.2.9 L'extraction des substitutions d'une séquence

Signature: $[SUBST : ASSC \longrightarrow AS]$

Déroulement concret

Etant donné une séquence programme munie de l'information sur le *cut*, cette opération retourne l'ensemble des substitutions programme contenues dans la séquence.

Spécification

Soient $C \in ASSC, S \in PSS$ *et* $cf \in CF$,

$< S, cf > \in \ Cc(C) \Longrightarrow Subst(S) \subseteq Cc(SUBST(C))$.

Implémentation

$\forall C \in ASSC$, nous avons:

$SUBST(C) = \beta$

Correction

C'est évident car β est la seule substitution abstraite abstrayant toutes les substitutions programmes⊙

1.2.10 L'extension du résultat d'un appel

Signature: $[EXTGS(l,.,.) : ASSC \times ASS \longrightarrow ASSC]$

Déroulement concret

Utilisant les séquences programmes obtenues par l'exécution d'une procédure appélée, cette opération met à jour la séquence munie de l'information sur le *cut* reçue avant l'appel. Plus précisement, lors de l'exécution du jème atome de la clause *tête* : $-a_1, \ldots, a_{j-1}, p(X_{i_1}, \ldots, X_{i_m}), a_{j+1}, \ldots, a_q$,
Cette opération procède en quatre étapes suivantes:

1. "Sauvegarder" $S_{in} =<< \theta_1, \theta_2, \ldots, \theta_n >, cf >$, la séquence munie de l'information sur le cut obtenue après les exécutions successives des atomes $a_1, \ldots, a_{j-1}$;
2. Pour chaque $\theta_i (1 \leq i \leq n)$, utiliser $RESTRG(\theta_i, a_j)$ et exécuter a_j afin d'obtenir $S'_i =< \theta'_{i_1}, \theta'_{i_2}, \ldots, \theta'_{i_{r_i}} >$
3. Calculer $S_i = EXTG(a_j, \theta_i, S'_i) =< \theta_i\sigma_1, \ldots, \theta_i\sigma_{i_{r_i}} >$ où les σ_j sont telles que $\theta_i\sigma_{j/\{X_{i_1},\ldots,X_{i_m}\}} = \theta'_j$;
4. Calculer la concaténation $S_{out} = S_1 :: S_2 :: \ldots :: S_n$.

Remarquons que si une S_i est infinie ou incomplète, toutes les S_j telles que $i < j \leq n$ seront simplement ignorées.

Définition [la concaténation,$\Box : ASS \times ASS \rightarrow ASS$]

Soient S_1 et S_2 deux séquences de substitutions. Nous définissons:

$$\begin{array}{lcll} S_1 \Box S_2 & = & S_1 & \text{si } S_1 \text{ est infinie ou incomplète} \\ & = & S_1 :: S_2 & \text{ailleurs} \end{array}$$

Suivant la même logique, nous définissons récursivement,

$$\begin{array}{lcll} \Box_{i=1}^{Ns} S_i & = & <> & \text{si } Ns = 0 \\ & = & S_1 & \text{si } Ns = 1 \\ & = & (S_1 \Box S_2) \Box (\Box_{i=3}^{Ns} S_i) & \text{ailleurs} \end{array}$$

Exemple

Nous référant à la figure 1.3, détaillons les calculs de la séquence S_3 obtenue par application de EXTGS sur S_2 et $q(X_2)$:

		Première étape
S_2	=	$< \theta_{2,1}, \theta_{2,2} >$
$\theta_{2,1}$	=	$\{X_1/a, X_2/y_1\}$
$\theta_{2,2}$	=	$\{X_1/b, X_2/y_1\}$
		Deuxième étape
$\theta'_{2,1}$	=	$RESTRG(q(X_2, \theta_{2,1})$
	=	$\{X_1/y_1\}$
$\theta'_{2,2}$	=	$RESTRG(q(X_2, \theta_{2,2})$
	=	$\{X_1/y_1\}$
$\langle \theta'_{2,1}, q \rangle$	$\mapsto$	$S'_{2,1}$
$S'_{2,1}$	=	$< \{X_1/a, X_2/b\}$
$\langle \theta'_{2,2}, q \rangle$	$\mapsto$	$S'_{2,2}$
$S'_{2,2}$	=	$< \{X_1/a, X_2/b\} >$
		Troisième étape
$S_{2,1}$	=	$EXTG(q(X_2, \theta_{2,1}, S'_{2,1})$
	=	$< \{X_1/a, X_2/a\}, \{X_1/a, X_2/b\} >$
$S_{2,2}$	=	$EXTG(q(X_2, \theta_{2,2}, S'_{2,2})$
	=	$< \{X_1/b, X_2/a\}, \{X_1/b, X_2/b\} >$
		Quatrième étape
S_3	=	$S_{2,1} \Box S_{2,2}$
	=	$< \{X_1/a, X_2/a\}, \{X_1/a, X_2/b\}, \{X_1/b, X_2/a\}, \{X_1/b, X_2/b\} >$

Spécification

Soient $C \in ASSC$, $B \in ASS$, $< S, cf > \in (PSS \times CF)$ *et* $S'_1, \ldots, S'_{Ns(S)} \in PSS$.

$$\left\{ \begin{array}{l} < S, cf > \in Cc(C), \\ S = < \theta_1, \ldots, \theta_{Ns(S)}, - >, \\ \{\forall k : 1 \leq k \leq Ns(S) : \\ S'_k \in Cc(B), \\ S_k = EXTG(l, \theta_k, S'_k)\} \end{array} \right\} \Longrightarrow < \Box_{k=1}^{N(s)} S_k, cf > \in Cc(EXTGS(l, C, B))$$

Implémentation

Soit B $\in$ ASS. Définissons $EXTGS_{aux}$ par les six (6) équations suivantes:

1. $EXTGS_{aux}(l, < BC, cf >, B) = \{< BC, cf >\}$
2. $EXTGS_{aux}(l, < 0, cf >, B) = \{< 0, cf >\}$
3. $EXTGS_{aux}(l, < 1, cf >, B) = B \times \{cf\}$
4. $EXTGS_{aux}(l, < 1^+, cf >, B) = ((B \cap \{BC, 1^+, 2^+\}) \cup \{inc(x)/x \in B\}) \times \{cf\}$
   où
$$inc(x) = \begin{cases} BC & \text{si } x = 0 \\ 1^+ & \text{si } x = 1 \\ 2^+ & \text{si } x = 2 \\ x & ailleurs \end{cases}$$
5. $$EXTGS_{aux}(l, < 2, cf >, B) = \bigcup_{b \in B} EXTGS_{aux}(l, < 2, cf >, b)$$
   où
$$EXTGS_{aux}(l, < 2, cf >, b) = \begin{cases} \{< b, cf >\} & \text{si } b \neq 1 \\ \{< 2, cf >\} & \text{si } b = 1 \end{cases}$$
6. $$EXTGS_{aux}(l, < 2^+, cf >, B) = \bigcup_{b \in B} EXTGS_{aux}(l, < 2^+, cf >, b)$$
   où
$$EXTGS_{aux}(l, < 2^+, cf >, b) = \begin{cases} \{< b, cf >\} & \text{si } b \in \{BC, 1^+, 2^+\} \\ \{< 2^+, cf >\} & \text{si } b \in \{1, 2\} \\ \{< BC, cf >\} & \text{si } b = 0 \end{cases}$$

Finalement, $\forall (C, B) \in (ASSC \times ASS)$

$$EXTGS(l, C, B) = \bigcup_{c \in C} (EXTGS_{aux}(l, < c, cf >, B)$$

Notation

Nous noterons dans la suite $S =< \theta_1, \ldots, \theta_n, * >$ pour signifier une séquence infinie ou incomplète

Correction de $EXTGS_{aux}$

Remarquons que les trois premiers cas sont évidents et que l'information sur le *cut*, cf, reste inchangée dans tous les cas. De ce fait, elle n'est pas reprise dans la démonstration.

- Si $c = 1^+$, nous avons S=$< \theta_1, \perp >$. Et donc

$$\square_{k=1}^{Ns(S)} S_k = \begin{cases} S_1 & \text{si } S_1 \text{ est incomplète ou infinie} \\ S_1 ::< \perp > & \text{si } S_1 \text{ est finie} \end{cases}$$

  En d'autres termes, les séquences infinies ou incomplètes sont restées intactes tandis que les séquences finies sont transformées en des séquences infinies exactement suivant la logique de la formule 4 de $EXTGS_{aux}$ ci-dessus.

  Plus précisément, si S_1 est infinie, les hypothèses $S_1' \in Cc(B)$ et $S_1 = EXTG(l, \theta_1, S_1')$ confirment qu'il existe $b \in (\{BC, 1^+, 2^+\} \cap B)$ tel que $S_1 \in Cc(b)$. Et donc: $\square_{k=1}^{Ns(S)} S_k = S_1 \in Cc(\{BC, 1^+, 2^+\} \cap B)$.

  Par ailleurs, si S_1 est finie, les hypothèses $S_1' \in Cc(B)$ et $S_1 = EXTG(l, \theta_1, S_1')$ confirment qu'il existe $b \in (\{0, 1, 2\} \cap B)$ tel que $S_1 \in Cc(b)$. Et donc: $\square_{k=1}^{Ns(S)} S_k = S_1 ::< \perp > \in Cc(inc(b)) \subseteq Cc(\{inc(x) : x \in B\})$.

- Si $c = 2$, traitons toutes les possibilités dans le tableau suivant:

2	**b**	**concaténation concrète**	**Résultat**
	BC	$< \perp > \square < \perp > \ldots \square < \perp >=< \perp >$	BC
	0	$< > \square < > \ldots \square < >=< >$	0
	1	$< \theta_1 > \square < \theta_2 > \ldots \square < \theta_n >=< \theta_1, \ldots, \theta_n >$	2
	1^+	$< \theta_1, \perp > \square < \theta_2, \perp > \ldots < \theta_n, \perp >=< \theta_1, \perp >$	1^+
	2	$< \theta_1, \ldots, \theta_n > \square < \theta_{n+1}, \ldots, \theta_m >$ $\ldots \square < \theta_{m+p}, \ldots, \theta_{m+p+q} >=< \theta_1, \ldots, \theta_{m+p+q} >$	2
	2^+	$< \theta_1, \ldots, \theta_n, * > \square < \theta_{n+1}, \ldots, \theta_{m+p+q}, * >$ $\ldots \square < \theta_{m+p}, \ldots, \theta_{m+p+q}, * >=< \theta_1, \ldots, \theta_n, * >$	2^+

Remarquons que les résultats à la dernière colonne du tableau ci-dessus correspondent bien à la définition (5) de $EXTGS_{aux}$.

- Si $c = 2^+$, traitons toutes les possibilités dans le tableau suivant:

2^+	b	concaténation concrète	Résultat
	BC	$< \perp > \square < \perp > \ldots \square < \perp > \ldots = < \perp >$	BC
	0	$< > \square < > \ldots \square < > \ldots = < \perp >$	BC
	1	$< \theta_1 > \square < \theta_2 > \ldots \square < \theta_n > \ldots = < \theta_1, \ldots, \theta_n, * >$	2^+
	1^+	$< \theta_1, \perp > \square < \theta_2, \perp > \ldots < \theta_n, \perp > \ldots = < \theta_1, \perp >$	1^+
	2	$< \theta_1, \ldots, \theta_n > \square < \theta_{n+1}, \ldots, \theta_m >$ $\ldots \square < \theta_{m+p}, \ldots, \theta_{m+p+q} > \ldots = < \theta_1, \ldots, \theta_{m+p+q}, * >$	2^+
	2^+	$< \theta_1, \ldots, \theta_n, * > \square < \theta_{n+1}, \ldots, \theta_m, * >$ $\ldots \square < \theta_{m+p}, \ldots, \theta_{m+p+q}, * > \ldots = < \theta_1, \ldots, \theta_n, * >$	2^+

Comme pour le cas précédent, nous remarquons que les résultats à la dernière colonne du tableau ci-dessus correspondent bien à la définition (6) de $EXTGS_{aux}\odot$.

Correction de EXTGS

Considérons $< S, cf > \in Cc(C)$ et $S'_k \in Cc(B)$. Par la définition de Cc, il existe $< b, cf > \in C$ et $b' \in B$ tels que $S \in Cc(< b, cf >)$ et $S'_k \in Cc(b')$. De l'hypothèse $S_k = EXTG(l, \theta_k, S'_k)$, de la correction de $EXTGS_{aux}$ et de la définition de Cc nous avons:
$< \square_{k=1}^{Ns(S)} S_k, cf > \in Cc(EXTGS_{aux}(l, < b, cf >, \{b'\})) \subseteq Cc(EXTGS(l, C, B))\odot$

1.2.11 La concaténation des résultats des clauses

Signature: $[CONC : ASSC \times ASS \longrightarrow ASS]$

Cette opération concatène le résultat de l'exécution d'une clause à celui des clauses suivantes. Toutefois, si la clause courante boucle, ou éventuellement contient un *cut* et donne une solution, les résultats de ses successeurs sont simplement ignorés.

Définition [La concaténation $\square : (ASS \times CF) \times ASS \rightarrow ASS$]

Soient $< S_1, cf > \in (ASS \times CF)$ et $S_2 \in ASS$. Nous définissons:

$$\begin{aligned} < S_1, cf > \square S_2 &= S_1 && \text{si } cf = cut \\ &= S_1 :: S_1 && \text{sinon} \end{aligned}$$

Exemple

Pour l'exemple de la figure 1.2, le résultat final découlant de la concaténation des résultats de deux clauses de p est décrit de la manière suivante:

$$\begin{aligned} S &= < S_5, cf_5 > \Box S_8 \\ &= < S_5, cut > \Box S_8 \\ &= S_5 \\ &= < \{X_1/a, X_2/a\} > \end{aligned}$$

Spécification

Soient $C \in ASSC$, $B \in ASS$, $\theta \in PSS$, $< S_1, cf > \in (PSS \times CF)$ *et* $S_2 \in PSS$:

$$\left\{ \begin{array}{l} < S, cf > \in Cc(C), \\ S_2 \in Cc(B), \\ \forall \theta' \in Subst(S_1) \cup Subst(S_2) : \theta' \leq \theta \end{array} \right\} \Longrightarrow < S_1, cf > \Box S_2 \in Cc(CONC(C, B))$$

Implémentation

Définissons l'opération $CONC_{aux}$ de la manière suivante:

1. Pour une séquence atomique munie de *cut*

   $\forall\, b, b' \in AASS$

   $$CONC_{aux}(< b, cut >, b' >) = b$$

2. Pour une séquence atomique munie de *nocut*

$CONC_{aux}$	BC	0	1	1^+	2	2^+
$< BC, nocut >$	BC	BC	BC	BC	BC	BC
$< 0, nocut >$	BC	0	1	1^+	2	2^+
$< 1, nocut >$	1^+	1	2	2^+	2	2^+
$< 1^+, nocut >$	1^+	1^+	1^+	1^+	1^+	1^+
$< 2, nocut >$	2^+	2	2	2^+	2	2^+
$< 2^+, nocut >$	2^+	2^+	2^+	2^+	2^+	2^+

Ainsi, $\forall C \in ASSC$ et $B \in ASS$,

$$CONC(C,B) = \bigcup_{<b,cf>\in C,\ b'\in B} CONC(< b,cf >, b')$$

Correction de $CONC_{aux}$ et $CONC$

Evidente par la définition de $< S, cf > \square S_2 \odot$

Chapitre 2

DOMAINE ABSTRAIT GENERIQUE

Nous allons maintenant reprendre presque le même travail en ajoutant un nouveau paramètre dans les séquences de substitutions abstraites. Une séquence de substitutions abstraites devient un couple $< \beta, L >$ où β est la substitution abstraite abstrayant toutes les substitutions programmes de la séquence programme considérée et $L \subseteq \wp(AASS)$ renseigne sur les différentes longueurs de la même séquence programme.

Etant donné que le déroulement des opérations concrètes est indépendant du domaine abstrait, nous ne le rappelons pas dans ce chapitre. Par ailleurs, si pour la plupart des opérations abstraites les spécifications restent inchangées, il n'en est pas de même pour quelques-unes. Afin d'éviter toute confusion et surtout d'avoir à l'esprit et devant les yeux le "comment implémenter une opération donnée", nous nous sommes obligés de rappeller ou de détailler les spécifications de chacune des opérations abstraites traitées et de recopier, si nécessaire, les résultats du chapitre précédent.

2.1 Domaine Abstrait

2.1.1 La concrétisation d'une séquence abstraite

Dans ce domaine, la concrétisation d'une séquence est définie de la manière suivante: $Cc(< \beta, L >) = Seq(\beta) \cap Cc(L)$
où
$Seq(\beta) = \{S \in PSS : Subst(S) \subseteq Cc(\beta)\}$
et $Cc(L) = \{S \in PSS : (\exists b \in L) \wedge (S \in Cc(b))\}$

2.1.2 Séquences Abstraites avec information sur le *cut*

L'ensemble des séquences abstraites avec information sur le *cut* devient: $ASSC = AS \times \wp(AASS \times CF)$. Pour une séquence $C =< \beta, LC >\in ASSC$, nous définissons

$$Cc(C) = Seq(\beta) \cap \bigcup_{<b,cf>\in LC} Cc(< b, cf >)$$

où $Cc(< b, cf >) = \{< S, cf >: S \in Cc(b)\}$.

2.2 Opérations Abstraites

2.2.1 L'entrée d'une clause

Signature: $[EXTC(c, .) : AS \longrightarrow ASSC]$

Spécification

Soient $\beta \in AS$, $\theta \in PSS$ (une substitution à n variables) et c une clause à m variables (avec $n \leq m$).

$$\theta \in Cc(\beta) \Longrightarrow EXTC(c, \theta) \in Cc(EXTC(c, \beta)).$$

L'implémentation de $EXTC$ utilise une opération auxiliaire, $EXTC_aux$, que nous spécifions de la manière suivante:
$\beta' = EXTC_{aux}(c, \beta) \Longrightarrow EXTC(c, \theta) \in Seq(\beta')$.

Implémentation

$$EXTC(c,\beta) =< EXTC_{aux}(\beta), \{< 1, nocut >\} >$$

Correction

Soit $\theta' \in PSS$ telle que $x_i\theta' = x_i\theta$ $(\forall\, i : 1 \leq i \leq n)$ et $x_{n+1}\theta', \ldots, x_m\theta'$ des variables standards distinctes non contenues dans $codom(\theta)$.

Nous avons:

$$\begin{array}{rcl} EXTC(c,\theta) & = & << \theta' >, nocut > \\ & \in & Seq(\beta') \cap Cc(\{< 1, nocut >\}) \\ & = & Cc(< \beta', \{< 1, nocut >\} >) \\ & = & Cc(EXTC(c,\beta))\odot \end{array}$$

2.2.2 La sortie d'une clause

Signature: $[RESTRC(c,.) : ASSC \longrightarrow ASSC]$

Spécification

Soient $C =< \beta, LC > \in ASSC$ et $< S, cf > \in (PSS \times CF)$.

$$< S, cf > \in Cc(C) \Longrightarrow RESTRC(c, < S, cf >) \in Cc(RESTRC(c, C)).$$

L'implémentation de $RESTRC$ utilise une opération supplémentaire, $RESTRC_{aux}$, que nous spécifions par:

$$RESTRC_{aux}(c,\beta) = \beta' \Longrightarrow RESTRC(c,S) \in Seq(\beta').$$

Implémentation

$$RESTRC(c,C) =< RESTRC_{aux}(c,\beta), \bigcup_{<b,cf>\in LC} RESTRC_{suppl}(< b, cf >) >$$

où
$RESTRC_{suppl}(< b, cf >) =< b, cf >$

Correction de $RESTRC$

Evidente par construction et la correction énoncée au point 1.2.3$\odot$

2.2.3 La préparation à un appel

Signature: $[RESTRG(l,.) : AS \longrightarrow AS]$

Spécification

Soient $\beta \in AS$ et $\theta = \{X_1/t_1, \ldots, X_n/t_n\} \in PSS$ avec $\theta \in Cc(\beta)$ et $l = q(X_{i_1}, \ldots, X_{i_k})$ où $k \leq n$.

$\theta \in Cc(\beta) \Longrightarrow RESTRG(l,\theta) \in Cc(RESTRG(l,\beta))$.

L'implémentation de $RESTRG$ exige une opération auxiliaire, $RESTRG_{aux}$, que nous spécifions de la manière suivante:

$RESTRG_{aux}(l,\beta) = \beta' \Longrightarrow \{X_1/t_{i_1}, \ldots, X_k/t_{i_k}\} \in Cc(\beta')$

Implémentation

$RESTRG(l,\beta) = \beta'$

Correction

Par la spécification de $RESTRG_{aux}$, nous obtenons trivialement $RESTRG(l,\theta) = \{X_1/t_{i_1}, \ldots, X_k/t_{i_k}\} \in Cc(\beta') = RESTRG(l,\beta)\odot$

2.2.4 L'unification de deux variables

Signature: $[UNIF\_VAR : AS \longrightarrow ASS]$

Spécification

Soient $\beta \in AS$ $\theta \in PSS$.

$\theta \in Cc(\beta) \Longrightarrow UNIF\_VAR(X_1 = X_2, \theta) \in Cc(UNIF\_VAR(\beta))$.

L'implémentation de $UNIF\_VAR$ utilise l'opération $UNIF\_VAR_{aux}$ dont la spécification est décrite de la manière suivante:

Soient ss (*sure success*) et sf (*sure failure*) deux variables booléennes et SS l'ensemble des substitutions standards.

$< \beta', ss, sf >= UNIF\_VAR_{aux}(X_1 = X_2, \beta)$

$$\Longrightarrow \left\{ \begin{array}{l} 1.\forall \theta \in Cc(\beta) : \forall \sigma \in SS : \sigma \text{ unifie } X_1\theta \text{ et } X_2\theta \Longrightarrow \theta'\sigma \in Cc(\beta')\}; \\ 2.ss \Longrightarrow \forall \theta \in Cc(\beta) : X_1\theta \textit{ et } X_2\theta \text{ sont unifiables}; \\ 3.sf \Longrightarrow \forall \theta \in Cc(\beta) : X_1\theta \textit{ et } X_2\theta \text{ ne sont pas unifiables.} \end{array} \right\}.$$

Implémentation

Soit $< \beta', ss, sf >= UNIF\_VAR_{aux}(X_1 = X_2, \beta)$. Nous implémentons

$$\begin{aligned} UNIF\_VAR(X_1 = X_2, \beta) &= < \beta', \{1\} > \text{ si ss} \\ &= < \beta', \{0\} > \text{ si sf} \\ &= < \beta', \{0, 1\} > \text{ ailleurs} \end{aligned}$$

Correction

- Supposons ss et $\neg sf$ et considérons $\theta \in Cc(\beta)$. Par la spécification (2) de $UNIF\_VAR_{aux}$, nous savons que $X_1\theta$ *et* $X_2\theta$ sont unifiables. Par conséquent, $UNIF\_VAR(X_i = X_j, \theta) =< \theta'\sigma >$ où σ unifie $X_1\theta$ *et* $X_2\theta$. Comme la spécification (1) de $UNIF\_VAR_{aux}$ confirme que $\theta\sigma \in Cc(\beta')$, nous avons finalement:
  $< \theta\sigma > \in Cc(< \beta', \{1\} >) = UNIF\_VAR(X_i = X_j, \beta)$.

- Supposons maintenant sf et $\neg ss$ et considérons $\theta \in Cc(\beta)$. Par la spécification (3) de $UNIF\_VAR_{aux}$, nous savons que $X_1\theta$ *et* $X_2\theta$ ne sont pas unifiables. Et donc
  $<> \in Cc(< \beta', \{0\} >) = UNIF\_VAR(X_i = X_j, \beta)$.

- Supposons $\neg ss$ et $\neg sf$ et considérons $\theta \in Cc(\beta)$. Par la spécification de $UNIF\_VAR_{aux}$, nous ne savons rien dire de l'unification de $X_1\theta$ *et* $X_2\theta$. Toutefois, que $UNIF\_VAR(X_i = X_j, \theta) =< \theta'\sigma >$ où σ unifie $X_1\theta$ *et* $X_2\theta$ ou que $UNIF\_VAR(X_i = X_j, \theta) =< >$, nous confirmons, grâce aux deux cas précédents que,
  $UNIF\_VAR(X_i = X_j, \theta) \in Cc(< \beta', \{0, 1\} >)$
  $= UNIF\_VAR(X_i = X_j, \beta)$.

- Supposons enfin ss et sf à la fois. Dans ces conditions, $Cc(\beta)$ est obligatoirement vide sinon il existerait une substitution programme θ telle que $X_1\theta$ *et* $X_2\theta$ seraient à la fois unifiables et non unifiables. Et donc la proposition "$\theta \in Cc(\beta)$" étant fausse, nous avons:
  $\theta \in Cc(\beta) \Longrightarrow UNIF\_VAR(X_i = X_j, \theta) \in Cc(UNIF\_VAR(\beta))\odot$

2.2.5 L'unification d'une variable et d'un foncteur

Signature: $[UNIF\_FUNC(X_1 = f(X_2, \ldots, X_n), .) : AS \longrightarrow ASS]$

Spécification

Soient $\beta \in AS$ *et* $\theta \in PSS$,

$\theta \in Cc(\beta) \Longrightarrow UNIF\_FUNC(X_1 = f(X_2, \ldots, X_n), \theta) \in Cc(UNIF\_FUNC(\beta))$.

L'implémentation de $UNIF\_FUNC$ utilise l'opération $UNIF\_FUNC_{aux}$ dont la spécification est décrite de la manière suivante:

Soient ss (*sure success*) et sf (*sure failure*) deux variables booléennes et SS l'ensemble des substitutions standards.

$$< \beta', ss, sf >= UNIF\_FUNC_{aux}(X_1 = f(X_2, \ldots, X_n), \beta)$$

$$\Longrightarrow \left\{ \begin{array}{l} 1.\forall\theta \in Cc(\beta) : \forall\sigma \in SS : \sigma \text{ unifie } X_1\theta \text{ et } X_2\theta \Longrightarrow \theta'\sigma \in Cc(\beta')\}; \\ 2.ss \Longrightarrow \forall\theta \in Cc(\beta) : X_1\theta \textit{ et } f(X_2, \ldots, X_n)\theta \text{ sont unifiables et} \\ 3.sf \Longrightarrow \forall\theta \in Cc(\beta) : X_1\theta \textit{ et } f(X_2, \ldots, X_n)\theta \text{ ne sont pas unifiables.} \end{array} \right\}.$$

Implémentation

Soit $< \beta', ss, sf >= UNIF\_FUNC_{aux}(X_1 = f(X_2, \ldots, X_n), \beta)$.
Nous implémentons:

$$\begin{array}{rcl} UNIF\_FUNC(X_1 = f(X_2, \ldots, X_n), \beta) & = & < \beta', \{1\} > \text{si ss} \\ & = & < \beta', \{0\} > \text{si sf} \\ & = & < \beta', \{0, 1\} > \text{ailleurs} \end{array}$$

Correction

Même raisonnement que celle de l'$UNIF\_VAR\odot$

2.2.6 L'interprétation abstraite de cut

Signature: $[AI\_CUT : ASSC \longrightarrow ASSC]$

Spécification

Soient $S =< \theta_1, \ldots, \theta_n > \in PSS$, $cf \in CF$ et
$C =<< \beta, cf >, LC > \in ASSC$

$< S, cf > \in Cc(C) \Longrightarrow AI\_CUT(< S, cf >) \in AI\_CUT(C)$.

Implémentation

$\forall C =< \beta, LC > \in ASSC$,

$AI\_CUT(C) =< \beta, AI\_CUT(LC)^1 >$

Correction

- Si $cf = nocut$,

$$\begin{array}{rcl} AI\_CUT(< S, cf >) & = & < S, AI\_CUT(LC) > \\ & \in & Cc(C) \end{array}$$

par hypothèse et par la correction de 1.2.7

- Si $cf = cut$,

$$\begin{array}{rcl} AI\_CUT(< S, cf >) & = & << \theta_1 >, AI\_CUT(LC) > \\ & \in & Cc(C) \end{array}$$

car par la définition de la concrétisation $\theta_1 \in Subst(S) \subseteq Cc(\beta)$ $\odot$

[1] Lire implémentation du 1.2.7

2.2.7 L'extraction de la séquence

Signature: $[SEQ : ASSC \longrightarrow ASS]$

Spécification

Soient $C =< \beta, LC > \in ASSC$, $S \in PSS$ *et* $cf \in CF$,

$< S, cf > \in Cc(C) \Longrightarrow S \in Cc(SEQ(C))$.

Implémentation

Soient $< b, cf > \in (AASS \times CF)$,

$SEQ(C) =< \beta, SEQ(LC)^2 >$

Correction

Evidente par la correction du 1.2.8⊙

2.2.8 L'extraction des substitutions d'une séquence

Signature: $[SUBST : ASSC \longrightarrow AS]$

Spécification

Soient $C =< \beta, LC > \in ASSC$, $S \in PSS$ *et* $cf \in CF$,

$< S, cf > \in Cc(C) \Longrightarrow Subst(S) \subseteq Cc(SUBST(C))$.

Implémentation

$\forall C \in ASSC$, on a :

$SUBST(C) = \beta$

Correction

$\forall < S, cf > \in Cc(< \beta, LC >)$, nous avons évidemment $Subst(S) \subseteq Cc(\beta)$ par la définition de la concrétisation d'une séquence abstraite munie de la information sur le *cut*⊙

[2]Lire implémentation du 1.2.8

2.2.9 L'extension du résultat d'un appel

Signature: $[EXTGS(l,.,.) : ASSC \times ASS \longrightarrow ASSC]$

Spécification

Soient $C =< \beta_C, LC > \in ASSC$, $B =< \beta_B, L > \in ASS$, $< S, cf > \in (PSS \times CF)$ *et* $S'_1, \ldots, S'_{Ns(S)} \in PSS$.

$$\left\{ \begin{array}{l} < S, cf > \in Cc(C), \\ S =< \theta_1, \ldots, \theta_{Ns(S)}, - >, \\ \{\forall k : 1 \leq k \leq Ns(S) : \\ S'_k \in Cc(B), \\ S_k = EXTG(l, \theta_k, S'_k)\} \end{array} \right\} \Longrightarrow < \square_{k=1}^{N(s)} S_k, cf > \in Cc(EXTGS(l, C, B))$$

L'implémentation de $EXTGS$ utilise une opération supplémentaire, $EXTGS_{aux}$ que nous spécifions par: Si $\theta \in Cc(\beta)$ et $\theta' \in Cc(\beta')$, alors

$$\beta'' = EXTGS_{aux}(l, \beta, \beta') \Longrightarrow \forall \sigma \text{ telle que } \theta\sigma_{/dom(\theta')} = \theta', \theta\sigma \in Cc(\beta'')$$

Implémentation

De l'implémentation sans paramètre β de EXTGS donnée au point 1.2.9, nous construisons:

$EXTGS(l, < \beta_C, LC >, < \beta_B, L >) =$
$< EXTGS_{aux}(l, \beta_C, \beta_B), EXTGS(l, LC, L)^3 >$

Correction de $EXTGS_{aux}$

Evidente par la correction du 1.2.9 et par la spécification de l'opération $EXTGS_{aux}\odot$

[3] Lire implémentation du 1.2.9

2.2.10 La concaténation des résultats des clauses

Signature: $[CONC : ASSC \times \wp(ASS) \longrightarrow ASS]$

Spécification

Soient $C \in ASSC$, $B \in ASS$, $\theta \in PSS$, $< S_1, cf > \in (PSS \times CF)$ *et* $S_2 \in PSS$.

$$\left\{ \begin{array}{l} < S, cf > \in Cc(C), \\ S_2 \in Cc(B), \\ \forall \theta' \in Subst(S_1) \cup Subst(S_2) : \theta' \leq \theta \end{array} \right\} \Longrightarrow < S_1, cf > \Box S_2 \in Cc(CONC(C, B))$$

L'implémentation de $CONC$ utilise trois opérations abstraites: $UNION$,$EXCLUSIVE$ et $CONC_{aux}$.Nous spécifions les deux premières avant d'implémenter la troisième.

Spécification de UNION

Soient β,β_1 et $\beta_2 \in AS$ où β_ϕ.

1. $\forall \beta_1, \beta_2 \in AS,\ Cc(\beta_1) \cup Cc(\beta_2) \subseteq Cc(UNION(\beta_1, \beta_2))$;
2. Si β_ϕ est une séquence abstraite telle que $Cc(\beta_\phi) = \phi$, $\forall \beta \in AS,\ UNION(\beta, \beta_\phi) = UNION(\beta_\phi, \beta) = \beta$.

Spécification de EXCLUSIVE

Soient $\beta, \beta_1, \beta_2 \in AS$
$EXCLUSIVE(\beta, \beta_1, \beta_2) \Longrightarrow$
$\neg(\exists \theta \in Cc(\beta), \theta_1 \in Cc(\beta_1), \theta_2 \in Cc(\beta_2), \sigma_1, \sigma_2 : \theta\sigma_1 = \theta_1 \wedge \theta\sigma_2 = \theta_2)$.

Implémentation de $CONC_{aux}$

$$CONC_{aux}(< \beta_1, \{< b_1, cf_1 >\} >, < \beta_2, \{b_2\} >)$$

$$= \left\{ \begin{array}{ll} < \beta_1, \{< b_1, cf_1 >\} > & \text{si } cf_1 = cut \\ < UNION(\beta_1, \beta_2), b_1 \oplus b_2 > & \text{sinon} \end{array} \right.$$

où $\oplus$ est définie par le tableau suivant:

b_1	b_2	$b_1 \oplus b_2$
BC	b_2	BC
1^+	b_2	1^+
2^+	b_2	2^+
0	b_2	b_2
1	BC	1^+
1	1	2
1	2	2
1	2^+	2^+
2	$0, 1, 2$	2
2	$BC, 1^+, 2^+$	2^+

Implémentation de $CONC$

Soient $C =< \beta_1, \{< b_{1_1}, cf_{1_1} >, \ldots, < b_{1_{n_1}}, cf_{1_{n_1}} >\} >$ et $B =< \beta_2, \{b_{2_1}, \ldots, b_{2_{n_2}}\} >$. En prenant,

$$\beta_{1_i} = \begin{cases} \beta_1 & \text{si } b_{1_i} \in \{1, 1^+, 2, 2^+\} \\ \beta_\phi & \text{ailleurs} \end{cases}$$

où β_ϕ est telle que $Cc(\beta_\phi) = \phi$, nous avons finalement:

$$CONC(\beta, C, B) =$$

$$\bigcup_{\substack{1 \le i \le n_1 \\ 1 \le j \le n_2 \\ \neg(EXCLUSIVE(\beta, \beta_{1_i}, \beta_{2_j}))}} CONC_{aux}(< \beta_{1_i}, \{< b_{1_i}, cf_{1_i}\} >, < \beta_{2_j}, \{b_{2_j}\} >)$$

Correction de $CONC_{aux}$

Soient $< S_1, cf_1 > \in Cc(< \beta_1, \{< b_1, cf_1 >\})$ et $S_2 \in Cc(< \beta_2, \{b_2\})$. Nous devons démontrer que
$< S_1, cf_1 > \Box S_2 \in Cc(CONC_{aux}(< \beta_1, \{< b_1, cf_1\} >, < \beta_2, \{b_2\} >))$. En effet,

- Si $cf_1 = cut$, c'est évident car $< S_1, cut > \Box S_2 = S_1$.
- Si $cf = nocut$, considérons toutes les possibilités de b_1 et b_2 dans le tableau suivant:

b_1	b_2	Concaténation concrète	
BC	b_2	$<< \perp >, nocut > \square S_2 =< \perp >$	BC
1^+	b_2	$<< \theta_1, \perp >, nocut > \square S_2 =< \theta_1, \perp >$	1^+
2^+	b_2	$<< \theta_1, \ldots, \theta_n, * >, nocut > \square S_2 =< \theta_1, \ldots, \perp, \theta_n, * >$	2^+
0	b_2	$<< >, nocut > \square S_2 = S_2$	b_2
1	BC	$<< \theta_1 >, nocut > \square < \perp >=< \theta_1, \perp >$	1^+
1	1	$<< \theta_1 >, nocut > \square < \theta_2 >=< \theta_1, \theta_2 >$	2
1	2	$<< \theta_1 >, nocut > \square < \theta_2, \ldots, \theta_n >=< \theta_1, \ldots, \theta_n >$	2
1	2^+	$<< \theta_1 >, nocut > \square < \theta_2, \ldots, \theta_n, * >=< \theta_1, \ldots, \theta_n, * >$	2^+
2	0	$<< \theta_1, \ldots, \theta_n >, nocut > \square < >=< \theta_1, \ldots, \theta_n >$	2
2	BC	$<< \theta_1, \ldots, \theta_n >, nocut > \square < \perp >=< \theta_1, \ldots, \theta_n, \perp >$	2^+
2	1	$<< \theta_1, \ldots, \theta_n >, nocut > \square < \theta_{n+1} >=< \theta_1, \ldots, \theta_{n+1} >$	2
2	1^+	$<< \theta_1, \ldots, \theta_n >, nocut > \square < \theta_{n+1}, \perp >=< \theta_1, \ldots, \theta_{n+1}, \perp >$	2^+
2	2	$<< \theta_1, \ldots, \theta_n >, nocut > \square < \theta_{n+1}, \ldots, \theta_m >=< \theta_{n+1}, \ldots, \theta_m >$	2
2	2^+	$<< \theta_1, \ldots, \theta_n >, nocut > \square < \theta_{n+1}, \ldots, \theta_m, * >=< \theta_1, \ldots, \theta_m, * >$	2^+

Remarquons que le résultat sur la dernière colonne du tableau ci-dessus correspond bien à l'implementation de $CONC_{aux}$

Correction de $CONC$

Elle découle de la définition de la concrétisation et de la correction de $CONC_{aux}\odot$

Chapitre 3

IMPLEMENTATION ET EVALUATION D'UNE INSTANCE DU DOMAINE GENERIQUE

Il s'agit dans ce chapitre d'instancier le premier paramètre, β, de nos séquences abstraites génériques afin d'implémenter (en CaML) l'algorithme d'interprétation abstraite [8] pour évaluer quelques procédures Prolog de manipulation des entiers naturels.

Le présent chapitre est subdivisé en cinq sections. La première spécifie l'instanciation du domaine abstrait.La deuxième donne une implémenta-tion de haut niveau des opérations abstraites auxiliaires relatives au paramètre générique β que nous nous sommes limités à spécifier dans le deuxième chapitre. La troisième présente brièvement l'algorithme d'interprétation abstraite que nous avons implémenté. La quatrième définit les types CaML représentants les éléments du domaine abstrait instancié et du Prolog pur. La cinquième et dernière section teste et évalue les procédures d'un programme Prolog de manipulation des entiers naturels.

Avant d'aborder la première section, signalons en passant que le type *entier naturel* ($\mathbb{N}$) peut-être représenté en Prolog par l'ensemble des termes

$$\mathbb{N}at = \{0, s(0), s(s(0)), \ldots, s(s(\ldots(s(0))\ldots)), \ldots\}.$$

3.1 Domaine abstrait et opérations abstraites

3.1.1 Séquences abstraites instanciées

Nous spécialisons la séquence abstraite générique $< \beta, L >$ en $<< \mu, \tau, \rho >, L >$ où :

- μ décrit les modes des termes associés aux variables et est de la forme $\{X_1/mo_1, \ldots, X_n/mo_n\}$ avec $(mo_i)_{1 \leq i \leq n} \in$ **MODES**, où

  $$MODES = \{ground(GG), variable(VV), nosharing(HH), any(AA)\}.$$

  Rappelons qu'un terme est de mode *ground* lorsqu'il ne contient que des littéraux constants (0 *et* s). Il est de mode *variable* lorsque sa représentation est une variable au sens de Prolog pur. Il sera de mode *nosharing* lorsqu'il est variable (au sens de Prolog pur) et n'est contenu dans aucun autre terme présent dans la substitution programme considérée. Un terme est dit *any* s'il est soit *ground* soit *variable* ou autre chose (un mélange de *grounds* et de *variables*).

- τ décrit les types des termes associés aux variables et est de la forme $\{X_1/ty_1, \ldots, X_n/ty_n\}$ avec $(ty_i)_{(1 \leq i \leq n)} \in$ **TYPES**, où

  $$TYPES = \{naturel(NN), quelconque(QQ)\}.$$

  Rappelons que les termes naturels sont des éléments de *nat*. Tous les autres termes que nous manipulerons seront dits *quelconques*.

- ρ renseigne sur les liaisons entre variables. Il contient des équations de la forme $X_i = 0$,$X_i = X_j$ et $X_i = s(X_j)$ avec $i \neq j$.

Notons ***I*Termes** l'ensemble de tous les termes, ***I*Equations** l'ensemble de toutes les équations pouvant être contenues dans un ρ, **vars(t)** l'ensemble de variables contenues dans le terme t, **var** un prédicat qui, appliqué à un terme t, retourne *vrai* si et seulement si le terme t est une variable et **equation** un prédicat qui, appliqué aux triplets (X_i, X_j, ρ), retourne *vrai* si et seulement si il existe dans ρ une équation liant X_i et X_j et définissons les différentes concrétisations suivantes:

3.1.2 Concrétisation des modes

La fonction $Cc : MODES \longrightarrow \wp(ITermes)$ est définie de la manière suivante:
$Cc(GG) = \{t : t \text{ est } ground\}$;
$Cc(VV) = \{t : var(t)\}$;
$Cc(HH) = \{t : var(t) \wedge \neg(\exists\, tt \in ITerm : t \in vars(tt))\}$[1]
$Cc(AA) = \{t : t \in ITermes\}$.

3.1.3 Concrétisation des types

La fonction $Cc : TYPES \longrightarrow \wp(ITermes)$ est définie de la manière suivante:
$Cc(NN) = \{t : t \in nat\}$;
$Cc(QQ) = \{t : t$ est un terme quelconque$\}$

3.1.4 Concrétisation des équations

Soit I une partie finie de $\mathbb{N}$ de la forme $\{1, 2, \ldots, n\}$. Nous définissons
La fonction $Cc_I : IEquations \longrightarrow \wp(ITermes)$ par:
$Cc_I(X_i = 0) = \{(X_j/t_j)_{j\in I} : t_i = 0\}$;
$Cc_I(X_i = X_k) = \{(X_j/t_j)_{j\in I} : t_i = t_k\}$
$Cc_I(X_i = s(X_k)) = \{(X_j/t_j)_{j\in I} : t_i = s(t_k)\}$

3.1.5 Concrétisation d'une substitution abstraite

Soit I l'ensemble d'indices d'une substitution $< \mu, \tau, \rho >$. La concrétisation $Cc : AS \longrightarrow \wp(PSS)$ est définie par: $Cc(< \mu, \tau, \rho >) = Cc(\mu) \cap Cc(\tau) \cap Cc(\rho)$ où
$Cc(\mu) = \{(X_i/t_i)_{i\in I} : t_i \in Cc(mo_i)\ \forall i \in I\}$
$Cc(\mu) = \{(X_i/t_i)_{i\in I} : t_i \in Cc(ty_i)\ \forall i \in I\}$
$Cc(\rho) = \bigcap_{eq\in\rho} Cc_I(eq)$

[1] Cette concrétisation est toujours calculée en fonction d'une substitution donnée.

3.2 Opérations abstraites

3.2.1 L'entrée d'une clause

Implémentation

Soient $X_1, \ldots, X_n$ les variables présentent dans la tête d'une clause c et $X_{n+1}, \ldots, X_m$ des nouvelles variables apparaissant dans le corps de c. Nous implémentons:

$EXTC_{aux}(c, < \mu, \tau, \rho >) =$
$< \mu \cup \{X_{n+1}/HH, \ldots, X_m/HH\}, \tau \cup \{X_{n+1}/QQ, \ldots, X_m/QQ\}, \rho >$

Correction

La justification découle du fait que $X_{n+1}\theta', \ldots, X_m\theta'$ sont des nouvelles variables non contenues dans le $codom(\theta)\odot$

3.2.2 La sortie d'une clause

Implémentation

Soit D l'ensemble de variables de la tête d'une clause c.

$$RESTRC_{aux}(c, < \mu, \tau, \rho >, L >) =< \mu_{/D}, \tau_{/D}, \rho_{/D} >$$

où

$\mu_{/D}$ et $\tau_{/D}$ sont obtenus en supprimant dans μ et τ les objets X_i/mo_ou_ty tel que X_i n'est pas reprise dans D. Tandis que $\rho_{/D}$ est obtenu en supprimant dans ρ_1 et ρ_2 toutes les équations dont l'un au moins des membres (variables) n'appartient pas à D.

Correction

La justification est évidente par construction$\odot$

3.2.3 La préparation à un appel

Implémentation

Soit l=p($X_{i_1}, \ldots, X_{i_k}$).

$$RESTRG(l, < \{X_1/mo_1, \ldots, X_n/mo_n\}, \{X_1/ty_1, \ldots, X_n/ty_n\}, \rho >) =$$
$$< \{X_1/mo_{i_1}, \ldots, X_k/mo_{i_k}\}, \{X_1/ty_{i_1}, \ldots, X_k/ty_{i_k}\}, \rho' >$$

où

$$\begin{array}{lcl} (X_r = 0) \in \rho' & \Longleftrightarrow & (X_{i_r} = 0) \in \rho \\ (X_r = s(0)) \in \rho' & \Longleftrightarrow & (X_{i_r} = s(0)) \in \rho \\ (X_r = X_j) \in \rho' & \Longleftrightarrow & (X_{i_r} = X_{i_j}) \in \rho \\ (X_r = s(X_j)) \in \rho' & \Longleftrightarrow & (X_{i_r} = X_{i_j}) \in \rho \end{array}$$

Correction

La justification est évidente car la même dénomination est utilisée au niveau concret⊙

3.2.4 L'unification de deux variables

Implémentation

l'implémentation de $UNIF\_VAR_{aux}$ utilise les opérations auxiliaires *changer_mod_GG*, *changer_mode_AA* , *changer_type* et *compare* que nous implémentons de la manière suivante:

$$Changer\_mode\_GG(X_i, X_j, < \mu, \tau, \rho >)$$
$$= \begin{cases} GG & \text{si } (mo_j = VV) \wedge (mo_i = GG) \wedge (equation(X_i, X_j, \rho)) \\ mo_j & \text{ailleurs} \end{cases}$$

$$Changer\_mode\_AA(X_i, X_j, < \mu, \tau, \rho >))$$
$$= \begin{cases} AA & \text{si } (mo_j = VV) \wedge (mo_i = AA) \wedge (equation(X_i, X_j, \rho) \\ mo_j & \text{ailleurs} \end{cases}$$

$$Changer\_type(X_i, X_j, < \mu, \tau, \rho >)$$
$$= \begin{cases} NN & \text{si } (ty_j = QQ) \wedge (mo_i = NN) \wedge (equation(X_i, X_j, \rho)) \\ mo_j & \text{ailleurs} \end{cases}$$

$compare(X_i, X_j, < \mu, \tau, \rho >)$

$$= \begin{cases} \{0\} \text{ si } & ((X_i = 0 \in \rho) \wedge (\exists k : X_j = s(X_k) \in \rho)) \vee \\ & ((X_j = 0 \in \rho) \wedge (\exists k : X_i = s(X_k) \in \rho)) \vee \\ & (X_j = s(X_i) \in \rho) \vee \\ & (X_i = s(X_j) \in \rho) \vee \\ \{1\} \text{ si } & ((X_i = 0 \in \rho) \wedge (X_j = 0 \in \rho)) \vee \\ & (X_j = s(X_i) \notin \rho) \wedge (X_i = s(X_j) \notin \rho) \end{cases}$$

Cela étant, nous implémentons $UNIF\_VAR_{aux}$ de la manière suivante:

$$UNIF\_VAR_{aux}(X_1 = X_2, < \{X_1/mo_1, \ldots, X_n/mo_n\}, \{X_1/ty_1, \ldots, X_n/ty_n\}, \rho >)$$
$$=< \{X_1/mo'_1, \ldots, X_n/mo'_n\}, \{X_1/ty'_1, \ldots, X_n/ty'_n\}, \rho > .$$

où $mo'_1, \ldots, mo'_n, ty'_1, \ldots, ty'_n$ et L' sont décrits dans le tableau suivant:

mo_1	mo_2	ty_1	ty_2	mo'_1	mo'_2	$mo'_3, \ldots, mo'_n$	ty'_1	ty'_2	$ty'_3, \ldots, ty'_n$	L'
GG	GG	NN	NN	-	-	-	-	-	-	$co(1,2)$
GG	HH	NN	QQ	–	GG	–	-	NN	-	$\{1\}$
GG	VV	NN	QQ	–	GG	$GG(i,2)$	-	NN	type(i,2)	$\{1\}$
GG	AA	NN	QQ	–	GG	$AA(i,2)$	-	NN	type(i,2)	$\{0,1\}$
HH	HH	QQ	QQ	VV	VV	–	-	-	-	$\{1\}$
HH	VV	QQ	QQ	VV	–	QQ	-	-	-	$\{1\}$
HH	AA	QQ	QQ	AA	–	–	-	-	-	$\{1\}$
VV	VV	QQ	QQ	–	–	–	-	-	-	$co(1,2)$
VV	AA	QQ	QQ	AA	–	$AA(i,2)$	-	-	-	$\{0,1\}$
AA	AA	QQ	QQ	–	–	$AA(i,1,2)$	-	-	-	$\{0,1\}$

Le contenu du tableau doit être interprété de la manière suivante:

1. la partie gauche du tableau définit les différents cas à considérer;
2. un tiret dans une des colonnes à droite veut dire que la valeur en sortie est identique à celle d'entrée:
3. $GG(i,2)$ signifie que $m'_i (3 \leq i \leq n)$ est trouvé après calcul de $Changer\_mode\_GG(X_2, X_i, < \mu, \tau, \rho >)$;
4. $AA(i,2)$ signifie que $m'_i (3 \leq i \leq n)$ est trouvé après calcul de $Changer\_mode\_AA(X_2, X_i, < \mu, \tau, \rho >)$;
5. $type(i,2)$ signifie que $m'_i (3 \leq i \leq n)$ est trouvé après calcul de $Changer\_type(X_2, X_i, < \mu, \tau, \rho >)$;

6. $AA(i,1,2)$ signifie que $m'_i(3 \leq i \leq n)$ est trouvé après calcul successifs de $Changer\_mode\_AA(X_1, X_i, < \mu, \tau, \rho >)$ et $Changer\_mode\_AA(X_2, X_i, < \mu, \tau, \rho >)$.

7. $co(i,j)$ signifie que L' est trouvé après calcul de $co(i, j, < \mu, \tau, \rho >)$;

Remarquons en passant que les six(6) autres cas ne sont pas considérés pour la simple raison que
$UNIF\_VAR(X_1 = X_2, < \{X_1/mo_1, \ldots, X_n/mo_n\}, \{X_1/ty_1, \ldots, X_n/ty_n\}, \rho >)$
$= UNIF\_VAR(X_2 = X_1, < \{X_1/mo_1, \ldots, X_n/mo_n\}, \{X_1/ty_1, \ldots, X_n/ty_n\}, \rho >)$. et que les variables booléennes ss et sf sont implicitement exprimées par L'. En effet, nous avons: $ss \iff L' = \{1\}$, $sf \iff L' = \{0\}$ et $((\neg ss) \wedge (\neg sf)) \iff L' = \{0,1\}$.

Correction

Soit $\beta =< \{X_1/mo_1, \ldots, X_n/mo_n\}, \{X_1/ty_1, \ldots, X_n/ty_n\}, \rho >$. Nous devons démontrer que si $\theta = \{X_1/t_1, \ldots, X_n/t_n\} \in Cc(\beta)$ et σ est un unificateur général de t_1 et t_2, alors $\theta\sigma = UNIF\_VAR(X_1 = X_2, \theta) \in Cc(UNIF\_VAR(X_1 = X_2, \beta))$. Pour y parvenir, nous traitons pas-à-pas les dix cas énoncés dans le tableau ci-dessus.

1. $mo_1 = GG \wedge mo_2 = GG \wedge ty_1 = NN \wedge ty_2 = NN$.

   Comme t_1 et t_2 sont *grounds*, qu'ils soient unifiables ou pas, nous avons trivialement $\theta = UNIF\_VAR(X_1 = X_2, \theta) \in \beta = Cc(UNIF\_VAR(X_1 = X_2, \beta))$. Si $t_1 = 0 = t_2$, $L' = \{1\}$. Si l'un des termes t_i $(i \in \{1,2\})$ est 0 et l'autre est de la forme $s(X_k)$ alors, $L' = \{0\}$.

2. $mo_1 = GG \wedge mo_2 = HH \wedge ty_1 = NN \wedge ty_2 = QQ$.

   Dans ces conditions, $t_1 = g_1$ est *ground* et $t_2 = X_k$ est une *variable* n'apparaissant pas dans les autres termes t_i. Et donc $L' = \{1\}$ car ss et que $\sigma = \{X_k/g_1\}$ unifie t_1 et t_2. Par conséquent,
   $\theta\sigma = \{X_1/g_1, X_2/g_1, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\}$
   $= \{X_1/g_1, X_2/g_1, X_3/t_3, \ldots, X_n/t_n\}$ car X_k est *nosharing* par hypothèse. Ceci correspond bien au résultat énoncé dans la deuxième ligne du tableau ci-dessus.

3. $mo_1 = GG \wedge mo_2 = VV \wedge ty_1 = NN \wedge ty_2 = QQ$. Dans ces conditions, $t_1 = g_1$ est *ground* et $t_2 = X_k$ est une *variable* pouvant apparaître dans les autres termes t_i. Et donc $L' = \{1\}$ car ss et que $\sigma = \{X_k/g_1\}$ unifie t_1 et t_2. Par conséquent, $\theta\sigma = \{X_1/g_1, X_2/g_1, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\}$ correspond bien au résultat énoncé dans la troisième ligne du tableau ci-dessus car

(a) si $mo_i = GG$ (et $ty_i = NN$), nous avons:
$mo'_i = GG = changer\_mode\_GG(X_i, X_2, \beta)$ et
$ty'_i = NN = changer\_type(X_i, X_2, \beta)$.

(b) si $mo_i = HH$ (et $ty_i = QQ$), nous avons:
$mo'_i = HH = changer\_mode\_GG(X_i, X_2, \beta)$ car t_i étant *nosharing*, est particulièrement différente de X_k et
$ty'_i = QQ = changer\_type(X_i, X_2, \beta)$.

(c) si $mo_i = VV$ (et $ty_i = QQ$), t_i est une *variable* qui peut ne pas être égale à X_k. Nous avons donc $t_i\sigma = t_i$ si $\neg(equation(X_k, t_i, \rho))$ et $t_i\sigma = g_1$ sinon. Ces résultats correspondent bien à
$mo'_i = changer\_mode\_GG(X_i, X_2, \beta)$ et
$ty'_i = changer\_type(X_i, X_2, \beta)$.

(d) si $mo_i = AA$ (et $ty_i = QQ$), t_i est *any* et donc $t_i\sigma$ le sera aussi. Ceci correspond bien à
$mo'_i = AA = changer\_mode\_GG(X_i, X_2, \beta)$ et
$ty'_i = QQ = changer\_type(X_i, X_2, \beta)$.

4. $mo_1 = GG \wedge mo_2 = AA \wedge ty_1 = NN \wedge ty_2 = QQ$. Dans ces conditions, $t_1 = g_1$ est *ground* et t_2 est *any*. Et donc $L' = \{0, 1\}$ car on ne sait rien de ss et de sf et $\theta\sigma = \{X_1/g_1, X_2/g_1, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\}$ correspond bien au résultat énoncé dans la quatrième ligne du tableau ci-dessus car

(a) si $mo_i = GG$ (et $ty_i = NN$), nous avons:
$mo'_i = GG = changer\_mode\_GG(X_i, X_2, \beta)$ et
$ty'_i = NN = changer\_type(X_i, X_2, \beta)$.

(b) si $mo_i = HH$ (et $ty_i = QQ$), nous avons:
$mo'_i = HH = changer\_mode\_GG(X_i, X_2, \beta)$ car t_i étant *nosharing*, n'apparaît particulièrement pas dans t_2 et
$ty'_i = QQ = changer\_type(X_i, X_2, \beta)$.

(c) si $mo_i = VV$ (et $ty_i = QQ$), t_i est une *variable* qui peut ne pas apparaître dans t_2. Nous avons donc $t_i\sigma = g_i$ si t_i apparaît dans t_2 ou $t_i\sigma = X_i$ (une *variable*) si t_i n'apparaît pas dans t_2. Les deux résultats correspondent bien à
$mo'_i = changer\_mode\_GG(X_i, X_2, \beta)$ et
$ty'_i = changer\_type(X_i, X_2, \beta)$.

(d) si $mo_i = AA$ (et $ty_i = QQ$), t_i est *any* et donc $t_i\sigma$ le sera aussi. Ceci correspond bien
$mo'_i = AA = changer\_mode\_GG(X_i, X_2, \beta)$ et
$ty'_i = QQ = changer\_type(X_i, X_2, \beta)$.

5. $mo_1 = HH \wedge mo_2 = HH \wedge ty_1 = QQ \wedge ty_2 = QQ$. Dans ces conditions, $t_1 = X_p$ et $t_2 = X_r$ sont des *variables* n'apparaissant pas dans les autres

termes t_i. Et donc $L' = \{1\}$ car ss est *true* et que $\sigma = \{X_r/X_p\}$ unifie t_1 et t_2. Par conséquent, $\theta\sigma = \{X_1/X_p, X_2/X_p, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\}$ $= \{X_1/X_p, X_2/X_p, X_3/t_3, \ldots, X_n/t_n\}$ car X_r est *nosharing* par hypothèse. Ceci correspond bien au résultat énoncé dans la cinquième ligne du tableau ci-dessus.

6. $mo_1 = HH \wedge mo_2 = VV \wedge ty_1 = QQ \wedge ty_2 = QQ$. Dans ces conditions, $t_1 = X_p$, est *nosharing* et $t_2 = X_r$ est une *variable* pouvant apparaître dans les autres termes t_i. Et donc $L' = \{1\}$ car ss est *true* et que $\sigma = \{X_p/X_r\}$ unifie t_1 et t_2. Par conséquent, X_p devient une *variable* et $\theta\sigma =$ $\{X_1/X_r, X_2/X_r, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\} = \{X_1/X_p, X_2/X_p, X_3/t_3, \ldots, X_n/t_n\}$ car X_p est *nosharing* par hypothèse. Ceci correspond bien au résultat énoncé dans la sixième ligne du tableau ci-dessus.

7. $mo_1 = HH \wedge mo_2 = AA \wedge ty_1 = QQ \wedge ty_2 = QQ$. Dans ces conditions, $t_1 = X_p$, est *nosharing* et t_2 est *any*. Et donc $L' = \{1\}$ car ss et que $\sigma = \{X_p/t_2\}$ unifie t_1 et t_2. Par conséquent, t_1 devient *any* et $\theta\sigma =$ $\{X_1/t_2, X_2/t_2, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\} = \{X_1/t_2, X_2/t_2, X_3/t_3, \ldots, X_n/t_n\}$ car X_p est *nosharing* par hypothèse. Ceci correspond bien au résultat énoncé dans la septième ligne du tableau ci-dessus.

8. $mo_1 = VV \wedge mo_2 = VV \wedge ty_1 = QQ \wedge ty_2 = QQ$. Dans ces conditions, t_1 $= X_p$ et $t_2 = X_r$ sont toutes deux des *variables* pouvant apparaître dans les autres termes t_i. Et donc $L' = \{1\}$ si $(\neg(X_1 = s(X_2) \wedge \neg(X_2 = s(X_1))$ou $L' = \{0\}$ si $((X_1 = s(X_2) \vee \neg(X_2 = s(X_1))$. Par ailleurs, $\theta\sigma =$ $\{X_1/X_r, X_2/X_r, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\} = \{X_1/X_p, X_2/X_p, X_3/t_3, \ldots, X_n/t_n\}$ car pour $\sigma = \{\}$ ou $\sigma = \{X_1/X_2\}$, on ne change rien en ce qui concerne les modes et les types de tous les termes. Ceci correspond bien au résultat énoncé dans la huitième ligne du tableau ci-dessus.

9. $mo_1 = VV \wedge mo_2 = AA \wedge ty_1 = QQ \wedge ty_2 = QQ$. Dans ces conditions, t_1 $= X_k$ est *variable* et t_2 est *any*. Et donc $L' = \{0, 1\}$ car on ne sait rien de ss et de sf et $\theta\sigma = \{X_1/t_2, X_2/t_2, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\}$ correspond bien au résultat énoncé dans la neuvième ligne du tableau ci-dessus car

   (a) si $mo_i = GG$ (et $ty_i = NN$), nous avons: $mo'_i = GG = changer - mode - AA(X_i, X_2, \beta)$ et $ty'_i = NN = ty_i$.

   (b) si $mo_i = HH$ (et $ty_i = QQ$), nous avons: $mo'_i = HH = changer - mode - AA(X_i, X_2, \beta)$ car t_i étant *nosharing*, n'apparaît particulièrement pas dans t_2 et $ty'_i = QQ = ty_i$.

   (c) si $mo_i = VV$ (et $ty_i = QQ$), t_i est une *variable* qui peut ne pas apparaître dans t_2. Nous avons donc $t_i\sigma = t_2$ si t_i est égale à X_k ou $t_i\sigma = t_i$ si t_i est différent de X_k . Les deux résultats correspondent bien à $mo'_i = changer - mode - AA(X_i, X_2, \beta)$ et $ty'_i = QQ = ty_i$.

(d) si $mo_i = AA$ (et $ty_i = QQ$), t_i est *any* et donc $t_i\sigma$ le sera aussi. Ceci correspond bien à $mo'_i = AA = changer\_mode\_GG(X_i, X_2, \beta)$ et $ty'_i = QQ = changer\_type(X_i, X_2, \beta)$.

10. $mo_1 = AA \wedge mo_2 = AA \wedge ty_1 = NN \wedge ty_2 = QQ$. Dans ces conditions, t_1 et t_2 sont *any*. Et donc $L' = \{0, 1\}$ car on ne sait rien de ss et de sf et $\theta\sigma = \{X_1/t_1\sigma, X_2/t_2\sigma, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\}$ où σ est de la forme $\{X_{i_1}/t_{i_1}, X_{i_2}/t_{i_1}, \ldots, X_{i_n}/t_{i_n}\}$. Seules les *variables* contenues dans t_1 et t_2 auront leur mode et leur type affectés par σ. Les autres termes restent intacts. Ceci correspond bien au résultat énoncé dans la dixième ligne du tableau ci-dessus⊙

3.2.5 L'unification d'une variable et d'un foncteur

Implémentation

SoientX_1 et X_2 deux variables quelconques. Relativement aux restrictions imposées à nos termes, les équations $X_1 = 0$ et $X_1 = s(X_2)$ sont les seules qui nécessitent l'opération $UNIF\_FUNC_{aux}$. L'implémentation de $UNIF\_FUNC_{aux}$ utilise les deux fonctions suivantes:

$$compare\_zero(X_i, < \mu, \tau, \rho >) = \begin{cases} \{0\} & \text{si } (\exists k : X_i = s(X_k) \in \rho) \\ \{1\} & \text{si } (X_i = 0 \in \rho) \end{cases}$$

1. Pour le premier cas, $UNIF\_FUNC_{aux}$ est définie par le tableau suivant:

mo_1	ty_1	mo'_1	$mo'_2, \ldots, mo'_n$	ty'_1	$ty'_2, \ldots, ty'_n$	L'
GG	NN	-	-	-	-	co_0(1)
HH	QQ	GG	-	NN	-	{1}
VV	QQ	GG	$GG(i,1)$	NN	type(i,1)	{1}
AA	QQ	GG	AA$(i,1)$	NN	-	{0,1}

où $co\_0(i)$ signifie le calcul de $compare\_zero(X_i, < \mu, \tau, \rho >)$

2. Pour le second cas, nous implémentons $UNIF\_FUNC_{aux}$ de la manière suivante:

$$UNIF\_FUNC_{aux}(X_1 = s(X_2), < \mu, \tau, \rho >) = \begin{cases} << \mu, \tau, \rho >, \{0\} > & \text{si } X_1 = X_2 \in \rho \\ UNIF\_VAR_{aux}(X_1 = X_2, < \mu, \tau, \rho >) & \text{ailleurs} \end{cases}$$

Correction

La justification est établie par un raisonnement analogue à celui de l'$UNIF\_VAR_{aux}$⊙

3.2.6 L'interprétation abstraite de *cut*

Implémentation

De l'implémentation de $AI - CUT$ donnée au point 2.2.6, constatons que AI_CUT_{aux} est implicitement la fonction identité.

Correction

Triviale par construction⊙

3.2.7 L'extraction de la séquence

Implémentation

De l'implémentation de SEQ donnée au point 2.2.7, remarquons que SEQ_{aux} est implicitement la fonction identité.

Correction

Triviale par construction.⊙

3.2.8 L'extraction des substitutions d'une séquence

Implémentation

De l'implémentation de $SUBST$ donnée au point 2.2.8, nous avons aussi

$$SUBST_{aux}(<< \mu, \tau, \rho >, LC >) =< \mu, \tau, \rho >$$

Correction

Triviale par construction⊙

3.2.9 L'extension du résultat d'un appel

Implémentation

Soient $l = p(X_1, \ldots, X_k)$, $\mu = \{X_1/mo_1, \ldots, X_n/mo_n\}$, $\tau = \{X_1/ty_1, \ldots, X_n/ty_n\}$, $\mu' = \{X_1/mo'_1, \ldots, X_n/mo'_k\}$, $\tau' = \{X_1/ty'_1, \ldots, X_n/ty'_k\}$. Nous définissons:

$$EXTGS_{aux}(l, <\mu, \tau, \rho>, LC >, << \mu', \tau', \rho' >) =$$
$$< \{X_1/mo''_1, \ldots, X_n/mo''_n\}, \{X_1/ty''_1, \ldots, X_n/ty''_n\}, \rho'' >.$$

où $mo''_1, \ldots, mo''_n, ty''_1, \ldots, ty''_n, \rho''$ sont décrits de la manière suivante:

- $\rho'' = \rho$;
- $\forall 1 \leq j \leq k$
  - mo''_j

$$= \begin{cases} VV & \text{si } (mo'_j = HH) \wedge (mo_j \neq HH) \\ GG & \text{si } (mo_j = VV) \wedge (mo''_j = VV) \wedge (\exists p : (1 \leq p \leq k) \wedge (equation(X_j, X_p, \rho)) \\ AA & \text{si } (mo_j = VV) \wedge (mo''_j = AA) \wedge (\exists p : (1 \leq p \leq k) \wedge (equation(X_j, X_p, \rho)) \\ mo'_j & ailleurs \end{cases}$$

  - ty''_j

$$= \begin{cases} QQ & \text{si } (ty'_j = QQ) \wedge (ty_j = QQ) \\ NN & \text{si } (ty_j = QQ) \wedge (ty''_j = QQ) \wedge (\exists p : (1 \leq p \leq k) \wedge (equation(X_j, X_p, \rho)) \\ NN & \text{si } (ty_j = NN) \vee (ty_j = NN) \end{cases}$$

- $\forall (k+1) \leq j \leq n$
  - mo''_j

$$= \begin{cases} GG & \text{si } (mo_j = VV) \wedge (\exists p : (1 \leq p \leq k) \wedge \\ & (mo''_p = GG) \wedge (equation(X_j, X_p, \rho)) \\ AA & \text{si } (mo_j = VV) \wedge (\exists p : (1 \leq p \leq k) \wedge \\ & (mo''_p = AA) \wedge (equation(X_j, X_p, \rho)) \\ mo_j & ailleurs \end{cases}$$

  - ty''_j

$$= \begin{cases} NN & \text{si } (ty_j = QQ) \wedge (ty''_j = QQ) \wedge (\exists p : (1 \leq p \leq k) \wedge \\ & (mo''_p = NN) \wedge (equation(X_j, X_p, \rho)) \\ ty_j & \text{ailleurs} \end{cases}$$

3.2.10 L'union

Implémentation

Soient $\mu = \{X_1/mo_1, \ldots, X_n/mo_n\}$,
$\tau = \{X_1/ty_1, \ldots, X_n/ty_n\}$,
$\mu' = \{X_1/mo'_1, \ldots, X_n/mo'_k\}$,
$\tau' = \{X_1/ty'_1, \ldots, X_n/ty'_k\}$.
Nous définissons:

$$UNION(<\mu, \tau, \rho>, <\mu', \tau', \rho'>) =$$
$$<\{X_1/mo''_1, \ldots, X_n/mo''_k\}, \{X_1/ty''_1, \ldots, X_n/ty''_n\}, \rho \cup \rho'>$$

où les mo''_j et ty''_j sont décrits dans les tableaux suivants:

mo_j	mo'_j	mo''_j
GG	GG	GG
GG	HH	AA
GG	VV	AA
GG	AA	AA
HH	HH	HH
HH	VV	VV
HH	AA	AA
VV	VV	VV
VV	AA	AA
AA	AA	AA

Les six autres cas sont obtenus par commutativité

ty_j	ty'_j	ty''_j
NN	NN	NN
NN	QQ	QQ
QQ	NN	QQ
QQ	QQ	AA

Correction

Si β et β' sont deux substitutions quelconques, nous devons démontrer que $Cc(\beta) \cup Cc(\beta') \subseteq Cc(UNION(\beta_1, \beta_2)) = Cc(\beta'')$.

En effet, supposons par l'absurde qu'il existe $(t_i)_{i\in I} \in Cc(\beta) \cup Cc(\beta')$ ($\in \beta$, par exemple) et $(t_i)_{i\in I} \notin Cc(\beta'')$. Il existe donc $p \in I$ telle que mo_p et mo''_p sont incompatibles.

- Si $mo_p = GG$, mo''_p doit appartenir à $\{VV, HH\}$ pour pouvoir être incompatible avec mo_p. Dans ces conditions, le tableau ci-dessus nous renseigne que mo_p et mo'_p appartiennent à $\{VV, HH\}$. Ceci contredit le fait que $mo_p = GG$.
- Si $mo_p = HH$, mo''_p doit appartenir à $\{GG\}$ pour pouvoir être incompatible avec mo_p. Dans ces conditions, le tableau ci-dessus nous renseigne que mo_p et mo'_p appartiennent à $\{GG\}$. Ceci contredit le fait que $mo_p = HH$.
- Si $mo_p = VV$, mo''_p doit appartenir à $\{GG, HH\}$ pour pouvoir être incompatible avec mo_p. Dans ces conditions, le tableau ci-dessus nous renseigne que mo_p et mo'_p appartiennent à $\{GG, HH\}$. Ceci contredit le fait que $mo_p = VV$.
- Si $mo_p = AA$, il est impossible qu'il soit incompatible avec mo''_p.

Le même raisonnement peut être utilisé pour établir la correction des types⊙

3.2.11 L'exclusion

Implémentation

Soient $\beta =< \mu, \tau, \rho >$, $\beta_1 =< \mu_1, \tau_1, \rho_1 >$ et $\beta_2 =< \mu_2, \tau_2, \rho_2 >$

trois substitutions quelconques. Définissons:

- $var\_ground(\beta) = \{X_i : (mo_i = GG) \wedge (ty_i = NN) \text{ dans } \beta\}$
- $var\_zero(\beta) = \{X_i : X_i = 0 \in \rho\}$
- $var\_sup(\beta) = \{X_i : \exists j \text{ tel que } X_i = s(X_j) \in \rho\}$

Finalement, pour j et k deux éléments différents de $\{1, 2\}$, nous implémentons:

$$EXCLUSIVE(\beta, \beta_1, \beta_2)$$

$$\Longleftrightarrow \left\{ \begin{array}{c} \exists X_i : \\ \\ X_i \in var\_ground(\beta) \\ \wedge \\ X_i \in var\_zero(\beta_k) \\ \wedge \\ X_i \in var\_sup(\beta_j) \end{array} \right\}$$

Correction

Supposons $EXCLUSIVE(\beta, \beta_2, \beta_3)$ et admettons qu'il existe $\theta = \{X_1/t_1, \ldots, X_n/t_n\} \in Cc(\beta)$, $\theta_1 = \{X_1/t_{1_1}, \ldots, X_n/t_{1_n}\} \in Cc(\beta_1)$, $\theta_2 = \{X_1/t_{2_1}, \ldots, X_n/t_{2_n}\} \in Cc(\beta_2)$, σ_1 et $\sigma_2 \in SS$ telles que $\theta_1 = \theta\sigma_1$ et $\theta_2 = \theta\sigma_2$.

Par la définition, il existe X_i telle que t_i est *ground*, $X_i \in var\_zero(\beta_1)$ et $X_i \in var\_sup(\beta_2)$. Comme les termes *ground* sont laissés intacts par les substitutions standards, les égalités $\theta_1 = \theta\sigma_1$ et $\theta_2 = \theta\sigma_2$ font que $t_{i_1} = t_{i_2}$. Ceci est absurde car t_{i_1} est 0 et t_{i_2} est de la forme $s(X_k)\odot$

3.3 L'Algorithme d'Interprétation Abstraite

Cette section vise à présenter, sans détails ni démonstrations, les opérations et les procédures de l'algorithme d'interprétation abstraite donné à la figure 3.1 .

Nous demanderons donc aux lecteurs intéréssés de consulter [8] où les résultats relatifs à la correction et à la terminaison sont systématiquement décrits.

Pour atteindre notre objectif, nous découpons la présente section en quatre points dont le premier définit quelques terminologies, le deuxième et le troisième definissent les opérations supplémentaires utilisées dans l'algorithme et le quatrième donne un fonctionnement intuitif de l'algorithme d'interprétation abstraite.

3.3.1 Terminologies

Définition [Noeud, $< \beta, p >$]

Par un noeud, nous sous-entendons un couple $< \beta, p >$ où β est une substitution abstraite et p un *string* indiquant le nom d'une procédure Prolog. l'ensemble de tous les noeuds sera désigné par $\mathbb{N}oeuds$.

Définition [Noeud marqué, $<< \beta, p >, B >$]

Par un noeud marqué, nous entendons un couple formé d'un noeud et d'une séquence abstraite représentant le résultat de p pour l'entrée β. B est aussi appélée *marquage* du noeud $< \beta, p >$ et est noté $sat < \beta, p >$.

Définition [Comportement abstrait, *sat*]

Un comportement abstrait, *sat*, est un ensemble de noeuds marqués. Comme nous le verrons plus loin, chaque itération de l'algorithme d'interprétation abstraite vise à améliorer le *sat* de l'étape précédente de telle sorte qu'en sortie chaque noeud soit attaché à son meilleur marquage.

Définition [Graphe de dépendance, *dp*]

Un graphe de dépendance est un sous ensemble de $\mathbb{N}oeuds \times \wp(\mathbb{N}oeuds)$. Un couple $<< \beta, p >, \{< \beta_1, p_1 >, \ldots, < \beta_n, p_n >\} >$ dans *dp* porte l'information suivante: "le résultat du noeud $< \beta, p >$ dépend des résultats des noeuds $< \beta_i, p_i >$ $(1 \leq i \leq n)$". L'ensemble $\{< \beta_1, p_1 >, \ldots, < \beta_n, p_n >\}$ sera aussi noté $dp < \beta, p >$.

```
procedure solve(in ⟨β,p⟩; out sat,dp);
   begin sat := ∅; dp := ∅; solve_call(⟨β,p⟩,∅,sat,dp) end;

procedure solve_call(in ⟨β,p⟩,suspended; inout sat,dp);
   if ⟨β,p⟩ ∉ (dom(dp) ∪ suspended) then
   begin
      if ⟨β,p⟩ ∉ dom(sat) then EXTEND(⟨β,p⟩,sat);
      repeat
         EXT_DP(⟨β,p⟩,dp);
         solve_procedure(⟨β,proc(p)⟩,suspended ∪ {⟨β,p⟩},B,sat,dp);
         ADJUST(⟨β,p,B⟩,sat,modified);
         REMOVE_DP(modified,dp)
      until ⟨β,p⟩ ∈ dom(dp)
   end;

procedure solve_procedure(in ⟨β,pr⟩,suspended; out B; inout sat,dp);
   begin
      c pr := pr; solve_clause(⟨β,c⟩,suspended,C,sat,dp);
      if empty(pr) or cut_or_nt(C) then B := SEQ(C) else
      begin solve_procedure(⟨β,pr⟩,suspended,B,sat,dp); B := CONC(β,C,B) end
   end;

procedure solve_clause(in ⟨β,c⟩,suspended; out C; inout sat,dp);
   begin
      C := EXTC(c,β);
      for i := 1 to m with l1,...,lm body-of c do
         if li is ! then
            C := AI-CUT(C)
         else begin
            β' := RESTRG(li,SUBST(C));
            switch (li) of
               case xj = xk: B := UNIF_VAR(β')
               case xj = f(...): B := UNIF_FUNC(f,β')
               case p'(...):
                  solve_call(⟨β',p'⟩,suspended,sat,dp); B := sat⟨β',p'⟩;
                  if ⟨β,name(c)⟩ ∈ dom(dp) then
                     ADD_DP(⟨β,name(c)⟩,⟨β',p'⟩,dp)
               end;
            C := EXTG(li,C,B)
         end;
      C := RESTRC(c,C)
   end
```

Figure 3.1: L'algorithme générique d'interprétation abstraite.

Le domaine d'un graphe de dépendance dp, noté $dom(dp)$, est l'ensemble des origines des couples de dp, c'est-à-dire:

$$dom(dp) = \bigcup_{<<\beta,p>,lt>\in dp} \{<\beta,p>\}$$

et le codomaine, noté $codom(dp)$, est la réunion de toutes les extrémités des couples de dp, c'est-à-dire:

$$codom(dp) = \bigcup_{<<\beta,p>,lt>\in dp} lt.$$

Définition [Fermeture transitive, $trans\_dp$]

Si $<\beta,p>$ est un noeud du $dom(dp)$, la fermeture transitive de $<\beta,p>$, noté $trans\_dp(<\beta,p>,dp)$, le plus petit sous-ensemble de $codom(dp)$ vérifiant les deux conditions suivantes:

1. Si $<\beta',p'> \in dp <\beta,p>$ alors $<\beta',p'> \in trans\_dp(<\beta,p>,dp)$;
2. Si $<\beta',p'> \in dp <\beta,p>$, $<\beta',p'> \in dom(dp)$ et $<\beta'',p''> \in trans\_dp(<\beta,p>,dp)$, alors $<\beta'',p''> \in trans\_dp(<\beta,p>,dp)$.

3.3.2 Opérations sur les comportements abstraits

EXTEND

Cette opération permet d'initialiser un noeud dans un sat donné. Elle est définie par:

$EXTEND(<\beta,p>,sat)$

$$= \begin{cases} sat \cup \{<\beta,p>,B_\perp> & \text{si } <\beta,p> \notin dom(sat) \\ sat & \text{ailleurs} \end{cases}$$

où $B_\perp$ est la séquence abstraite contenant la plus faible information. Pour nous, elle est représentée par la séquence est $<\beta_\phi,\{BC\}>$

ADJUST

Cette opération est utilisée pour mettre à jour le marquage d'un noeud, et donc un *sat*, à la fin d'une itération de l'algorithme d'interprétation abstraite. Pour un noeud marqué $<< \beta, p >, lt > \in sat$, nous définissons:

$$ADJUST(<< \beta, p >, B >, sat)$$

$$= \{<< \beta, p >, B \nabla sat < \beta, p >>\} \cup (sat \setminus \{<< \beta, p >, lt >\})$$

Remarquons que si $< \beta, L >$ et $< \beta', L' >$ sont deux séquences abstraites,

$$< \beta, L > \nabla < \beta', L' > = < UNION(\beta, \beta'), L \nabla L' > .$$

MODIFIED

Appliquer à un noeud d'un *sat* à la fin d'une itération, cette opération retourne ϕ si le marquage du noeud est resté inchangé ou le singléton formé du noeud dans le cas contraire. Elle est définie mathématiquement par:

$$MODIFIED(<< \beta, p >, B^2, sat)$$

$$= \begin{cases} \phi & \text{si } B = sat < \beta, p > \\ \{< \beta, p >\} & \text{ailleurs} \end{cases}$$

3.3.3 Opérations sur les graphes de dépendance

REMOVE_DP

Cette opération est utilisée pour enlèver dans un graphe de dépendance tous les couples dont les origines seraient contenues dans la fermeture transitive de l'un des noeuds reçus en argument.

$$REMOVE\_DP(\{< \beta_i, p_i >, \ldots, < \beta_i, p_i >\}, dp)$$

$$= dp \setminus \{< \beta, p >: \exists i((1 \leq i \leq n) \wedge < \beta, p > \in trans\_dp(< \beta_i, p_i >, dp))\}$$

[2]B représente le nouveau marquage de $< \beta, p >$

EXT_DP

Cette opération insère un nouveau couple dans le graphe de dépendance. Elle est définie par:
$EXT\_DP(< \beta, p >, dp)$

$$= \begin{cases} dp & \text{si } < \beta, p > \in dom(dp) \\ dp \cup \{< \beta, p >\} & \text{ailleurs} \end{cases}$$

ADD_DP

Pour un couple $<< \beta, p >, lt > \in dp$, cette opération est utilisée pour ajouter un élément donné dans lt. Nous la définissons mathématiquement par:

$ADD\_DP(< \beta, p >, < \beta', p' >, dp)$

$$= << \beta, p >, lt \cup \{< \beta', p' >\} > \cup (dp \setminus \{<< \beta, p >, lt >\}).$$

3.3.4 Algorithme d'interprétation abstraite

L'algorithme d'interprétation abstraite donné à la figure 3.1 est un algorithme de point fixe qui, **intuitivement**, pour un noeud $< \beta, p >$ donné , démarre avec $sat_0 = EXTEND(< \beta, p >, \phi)$ puis procède en étapes itératives dont chacune, exploitant les opérations abstraites décrites dans ce travail, vise améliorer le *sat* de l'étape précédente.

L'algorithme s'arrêtera lorsque, pour une étape $(i+1)$, on trouve $sat_{i+1} = sat_i$. Dans ces conditions, sat_i définit le point fixe cherché et donc le *sens abstrait* de l'exécution de p pour l'entrée β.

3.4 Les types CaML des objets du domaine abstrait et de Prolog

3.4.1 Types représentants les objets abstraits

1. **Un terme** est soit zero(0), soit une variable -répérée par son indice- ou soit un successeur d'un terme. Ce que nous avons traduit en CaML en:

$$Terme = ZERO|\ VAR\ of\ int|\ s\ of\ Terme$$

2. **Le type** d'un terme est soit *naturel*(NN) ou *quelconque*(QQ). Ce que nous avons traduit en :

$$Ty = NN|\ QQ$$

3. **Les Modes.** Il y a quatre modes: *ground*(GG),*nosharing*(HH), *variable*(VV) et *any*(AA). C'est-à-dire:

$$Mo = GG|\ HH|\ VV|\ AA$$

4. **Une information sur le** *cut* est soit *cut*(CU) ou *nocut*(NC). C'est-à-dire:

$$Cf = CU|\ NC$$

5. **Les séquences Abstraites Atomiques.** Les éléments BC,0,1,1^+,2 et 2^+ ont été respectivement traduits en:

$$Aass = INF|\ A\_ZERO|\ UN\_S|\ UN\_P|\ DEUX|\ DEUX\_P$$

6. **Le composant mode d'une séquence abstraite** associe chaque variable de la séquence au mode de son terme correspondant. Ce que nous avons traduit en:

$$Modes = M\ of\ (int * Mo)list$$

7. **Le composant type d'une séquence abstraite** associe chaque variable de la séquence au type de son terme correspondant. Ce que nous avons traduit en:

$$types = T\ of\ (int * Ty)list$$

8. **Le composant liaison d'une séquence abstraite** renseigne sur les différentes interdépendances liant les variables. Au lieu d'implémenter les équations telles que décrites dans le troisième chapitre, nous avons opté pour la découpe de ce paramètre en deux listes. La première contient les couples (i, j) informant sur la liaison des variables X_i et X_j et la seconde contient des

couples (i, l) portant l'information suivante: les varables indexées par l ont leurs valeurs "inférieures" à celle de X_i. Ce changement de représentation ne modifie aucunement notre implémentation de haut niveau car il est facile d'établir une bijection entre les deux manières de représenter ce paramètre. Ainsi, nous avons en CaML:

$$Liaisons = L\ of\ ((int * int)list * (int * (int\ list))list)$$

9. **Une substitution abstraite** est soit vide -à concrétisation vide- ou un triplet formé du composant mode, du composant type et du composant liaison. Ce qui se traduit en:

$$Sub\_Abst = SA\_VIDE|S\_Ab\ of\ (Modes * Types * Liaisons)$$

10. **Une séquence abstraite** est un couple formé d'une substitution abstraite et d'un ensemble de séquences abstraites atomiques. Ce que nous avons traduit en:

$$Seq\_Abst = Se\_Ab\ of\ (Sub\_Abst * (Aaas\ list))$$

11. **Une séquences abstraite munie de l'information sur le** *cut* est un couple formé d'une substitution abstraite et d'un ensemble de couples dont les origines sont des séquences abstraites atomiques et les extrémités des informations sur le *cut*. Nous l'avons traduit en

$$Seq\_Abst\_Cut = Sec\_Ab\ of\ (Sub\_Abst * ((Aaas * Cf)list))$$

3.4.2 Types représentants les objets de Prolog pur

1. **Un programme** est une liste de procédures.

$$Programme = PG\ of\ (Procedure\ list)$$

2. **Une procédure** est un couple formé du nom de la procédure et de la liste de ses clauses.

$$Procedure = PR\ of\ string * (Clause\ list)$$

3. **Une clause** est un couple formé d'une tête et d'un corps.

$$Clause = CL\ of\ (Tete * Corps)$$

4. **Une tête** est couple formé du nom de la clause et de la liste des indices de ses variables.

$$Tete = TE\ of\ (string\ *\ (int\ list))$$

5. **Un corps** est soit vide ou soit une liste d'atomes.

$$Corps = VIDE \mid CO\ of\ (Atome\ list)$$

6. **Un atome** est soit une unification des variables, soit une unification d'une variable et d'un foncteur, soit un appel d'une procédure ou un Cut.

$$\begin{aligned} Atome = \ & UN\ of\ (Terme\ *\ Terme) \mid \\ & FU\ of\ (Terme\ *\ Terme) \mid \\ & AP\ of\ string\ *\ (int\ list) \mid \\ & CUT \end{aligned}$$

3.4.3 Traduction du programme Prolog à évaluer

Le programme Prolog normalisé

$$\begin{array}{lll} nat(X1) & :- & X1 = 0. \\ nat(X1) & :- & X1 = s(X2),\ nat(X2). \\ & & \\ natPlus(X1, X2, X3) & :- & nat(X3),\ natPlusAux(X1, X2, X3). \\ & & \\ natPlusAux(X1, X2, X3) & :- & X1 = 0,\ X2 = X3. \\ natPlusAux(X1, X2, X3) & :- & X1 = s(X4),\ X3 = s(X5),\ natPlusAux(X4, X2, X5). \\ & & \\ natMul(X1, X2, X3) & :- & natPlus(X1, X2, X4),\ natMulgga(X1, X2, X3). \\ & & \\ natMulgga(X1, X2, X3) & :- & X1 = 0,\ X3 = 0. \\ natMulgga(X1, X2, X3) & :- & X1 = s(X4),\ natMulgga(X4, X2, X5), \\ & & \quad natPlusAux(X2, X5, X3). \end{array}$$

est traduit en CaML (suivant les types énoncés ci-dessus) en

```
let nat=
    (PR(("nat",[
        (CL(TE("nat",[1]))(CO[(FU((VAR 1),ZERO))])));
        (CL(TE("nat",[1])),
            (CO[(FU((VAR 1),s(VAR 2)));(AP("nat",[2]))])))
    ])))
in
let natPlus=
    (PR(("natPlus",[
        (CL((TE("natPlus",[1;2;3])),
            (CO[(AP("nat",[3]));(AP("natPlusAux",[1;2;3]))])))
    ])))
in
let natPlusAux=
    (PR(("natPlusAux",[
        (CL((TE("natPlusAux",[1;2;3])),
            (CO[(FU((VAR 1),ZERO));(UN((VAR 2),(VAR 3)))])));
        (CL((TE("natPlusAux",[1;2;3])),
            (CO[(FU((VAR 1),s(VAR 4)));(FU((VAR 2),s(VAR 5)));
                    (AP("natPlusAux",[4;2;5]))])))
    ])))
in
let bf natMul=
    (PR(("natMul",[
        (CL((TE("natMul",[1;2;3])),
            (CO[(AP("natPlus",[1;2;4]));
                    (AP("natMulgga",[1;2;3]))])))
    ])))
in
let natMulgga=
    (PR(("natMulgga",[
        (CL((TE("natMulgga",[1;2;3])),
            (CO[(FU((VAR 1),ZERO)); (FU((VAR 3),ZERO))])));
        (CL((TE("natMulgga",[1;2;3])),
            (CO[(FU((VAR 1),s(VAR 4))); (AP("natMulgga",[4;2;5]));
                    (AP("natPlusAux",[2;5;3]))])))
    ])))
in
pgm=
    [nat;natPlus;natPlusAux;natMul;natMulgga];;
```

Figure 3.2: Programme à tester

3.5 Evaluations expérimentales

Les résultats que nous énonçons dans cette section ont été obtenus par l'exécution de l'algorithme d'interprétation abstraite 3.1 que nous avons implémenté en CaML. Le programme CaML correspondant est donné en annexe du présent travail.

Le temps d'analyse des procédures de ce programme étant trop petit, nous mesurons le temps de 60 exécutions successives en utilisant une fonction récursive qui exploite la fonction prédéfinie *value time* : $unit \longrightarrow float =< fun >$ se trouvant dans la librairie *sys* du CaML(0.74).

3.5.1 Test de la procédure "nat"

Substitution abstraite d'entrée	$< \{X_1/GG\}, \{X_1/NN\}, [(1, [])] >$
Séquence abstraite de sortie	$<< \{X_1/GG\}, \{X_1/NN\}, [(1, [])] >, \{UN\_S\} >$
Contenu du dp	$\{<< \{X_1/GG\}, nat >, \{< \{X_1/GG\}, nat >\} >\}$
Temps d'exécution	0.22
Nombre de noeuds dans sat	1
Nombre d'itérations	2

Ce résultat traduit bien les exécutions concrètes de la procédure *nat* avec de la directionnalité *ground* (élément de $\mathbb{N}at$) qui donnent une et une seule réponse.

Pour la suite de l'évaluation, nous n'indiqueront plus le composants *types* et *liaisons* de substitutions abstraites.

Substitution abstraite d'entrée	$\{X_1/VV\}$
Séquences abstraites de sortie	$< \{X_1/GG\}, \{DEUX\_P\} >$
Contenu du dp	$\{<< \{X_1/VV\}, nat >, \{< \{X_1/VV\}, nat >\} >\}$
Temps d'exécution	0.33
Nombre de noeuds dans sat	1
Nombre d'itérations	3

L'interprétation de ce tableau est que l'exécution de la procédure *nat* suivant la directionnalité *variable* donne n $(2 \leq n)$ solutions *grounds* et ne se termine pas. En insérant un *CUT*, à la fin de la première clause de la procédure *nat* et en gardant la même directionnalité, nous obtenons :

Substitution abstraite d'entrée	$\{X_1/VV\}$
Séquences abstraites de sortie	$< \{X_1/GG\}, \{UN\_S\} >$
Contenu du dp	$\{\}$
Temps d'exécution	0.05
Nombre de noeuds dans sat	1
Nombre d'itérations	1

Le dernier test de *nat* consiste en la directionnalité *any* qui donne les résultats suivants:

Substitution abstraite d'entrée	$\{X_1/AA\}$
Séquences abstraites de sortie	$< \{X_1/GG\}, \{A\_ZERO, UN\_S, DEUX\_P\} >$
Contenu du dp	$\{\}$
Temps d'exécution	0.49
Nombre de noeuds dans sat	1
Nombre d'itérations	4

Remarquons que la directionnalité *any* peut donner aucune, une ou plusieurs solutions.

3.5.2 Test de la procédure "natPlusAux"

Analysons la procédure *natPlusAux* suivant la directionnalité $(ground, ground, ground)$.

Substitution abstraite d'entrée	$\{X_1/GG, X_2/GG, X_3/GG\}$
Séquences abstraites de sortie	$< \{X_1/GG, X_2/GG, X_3/GG\}, \{UN\_S\} >$
Contenu du dp	$\{<< \{X_1/GG, X_2/GG, X_3/GG\}, natPlusAux >,$ $\{< \{X_1/GG, X_2/GG, X_3/GG\}, natPlusAux >\} >$
Temps d'exécution	0.66
Nombre de noeuds dans sat	1
Nombre d'itérations	1

Remarquons que l'exécution se termine et donne une seule solution.

3.5.3 Test de la procédure "natPlus"

Analysons la procédure *natPlus* suivant la directionnalité $(ground, var, ground)$.

Substitution abstraite d'entrée	$\{X_1/GG, X_2/VV, X_3/GG\}$
Séquences abstraites de sortie	$< \{X_1/GG, X_2/GG, X_3/GG\}, \{DEUX\} >$
Contenu du dp	$\{<< \{X_1/GG, X_2/VV, X_3/GG\}, natPlus >,$ $\{< \{X_1/GG, X_2/VV, X_3/GG\}, natPlusAux >\} >$
Temps d'exécution	1.54
Nombre de noeuds dans sat	4
Nombre d'itérations	3

Pour la directionnalité $(ground, var, ground)$, la procédure *natPlus* donne au moins deux solutions et se termine.

3.5.4 Test de la procédure "natMulgga"

Analysons la procédure *natMulgga* suivant la directionnalité $(any, ground, ground)$.

Substitution abstraite d'entrée	$\{X_1/AA, X_2/GG, X_3/GG\}$
Séquences abstraites de sortie	$< \{X_1/GG, X_2/GG, X_3/GG\}, \{A\_ZERO, UN_S\} >$
Contenu du dp	$\{<< \{X_1/GG, X_2/VV, X_3/GG\}, natMulgga >,$ $\{\} >$
Temps d'exécution	0.06
Nombre de noeuds dans sat	1
Nombre d'itérations	1

Pour la directionnalité $(any, var, ground)$, la procédure *natMulgga* donne au plus une solution et se termine.

3.5.5 Test de la procédure "natMul"

Analysons la procédure $natMulgga$ suivant la directionnalité (any, any, any).

Substitution abstraite d'entrée	$\{X_1/AA, X_2/AA, X_3/AA\}$
Séquences abstraites de sortie	$< \{X_1/GG, X_2/GG, X_3/GG\}, \{A\_ZERO, UN_S, DEUX\_P\} >$
Contenu du dp	$\{<< \{X_1/HH\}, nat >,$ $\{< \{X_1/VV\}, natPlusAux >, < \{X_1/AA\}, natPlusAux >\} >,$ $<< \{X_1/AA, X_2/AA, X_3/HH\}, natPlus >,$ $\{< \{X_1/AA, X_2/AA, X_3/GG\}, natPlusAux >,$ $< \{X_1/HH\}, nat >\} >,$ $<< \{X_1/AA, X_2/AA, X_3/AA\}, natMul >,$ $\{< \{X_1/AA, X_2/AA, X_3/AA\}, natMulgga >,$ $< \{X_1/AA, X_2/AA, X_3/HH\}, natPlus >\} >\}$
Temps d'exécution	2.3
Nombre de noeuds dans sat	7
Nombre d'itérations	3

Pour la directionnalité $(any, var, ground)$, la procédure $natMulgga$ donne au plus une solution et se termine.

CONCLUSION

A la lumière des résultats obtenus au troisième chapitre, nous sommes heureux de constater que notre investigation est allée au-delà de l'objectif que nous nous sommes assignés.

En effet, le premier chapitre définit un outil puissant pouvant permettre à un compilateur Prolog d'exhiber, en dehors des différentes réponses relatives à une question posée, le *nombre* possible de ces solutions.

En outre, le domaine générique traité au deuxième chapitre offre une multitude d'analyses statiques de programmes dans la mesure où chaque spécialisation de β renseigne sur un aspect particulier de traitement de programmes Prolog avec *cut*.

La restriction que nous nous sommes faites en ne traitant que des entiers naturels a fait que le paramètre τ n'apporte pas grand-chose à notre analyse car les naturels ne sont rien d'autres que les termes *ground*. Malgré la redondance, nous avons voulu laisser ce paramètre pour permettre son exploitation dans un domaine où les termes *ground* peuvent ne pas être du même type.

Par ailleurs, l'analyse est un peu alourdie par le codage - suivant les types CaML énoncés au troisième chapitre - d'éventuels programmes Prolog pur à traiter. Ce problème peut être résolu par l'utilisation des outils d'analyse syntaxique.

En résumé, nous suggérerons que ce travail puisse être continué par la généralisation du composant type et par l'ajout des outils syntaxiques qu'offre d'ailleurs CaML dans sa version 0.74 sous Windows 3.1, 3.11 et 95.

Bibliographie

[1] B. Le Charlier. Interprétation abstraite. Notes de cours, Institut d'Informatique (Namur), 1998.

[2] W.F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.

[4] B. Le Charlier. L'Analyse Statique des Programmes par Interprétation Abstraite. *Nouvelles de la Science et des Technologies*, 9(4):19–25, 1992.

[5] B. Le Charlier. *Abstract Interpretation and Finite Domain Symbolic Constraints*, pages 147–170. Number 910 in Lecture Notes in Computer Science. Springer-Verlag, March 1995.

[6] B. Le Charlier. Abstract Interpretation and Application to Interactive System Verification. In *Proceedings of the Third Eurographics Workshop on Design, Specification, Verification of Interactive Systems (DSV-IS'96)*, Namur, Belgium, June 1996. Springer-Verlag/Wien. (Invited Paper).

[7] B. Le Charlier, O. Degimbe, L. Michel, and P. Van Hentenryck. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In Cousot P. and all, editors, *Proc of the Third International Workshop on Static Analysis (WSA'93)*, number 724 in Lecture Notes in Computer Science, Padova, September 1993. Springer-Verlag.

[8] B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence-Based Abstract Interpretation of Prolog. Technical Report RR-97-001, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, January 1997.

[9] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.

[10] D. Salhin. Determinacy Analysis for Full Prolog. In *Proceedings of the 1991 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (*PEPM'91*)*, 1991.

[11] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge Mass., 1986.

Annexe A

Définitions de types et quelques fonctions de base

A.1 Définitions de types

```
type
(*Les termes*)
   Terme=ZERO|VAR of int | s of Terme
and
(*Les types *)
   Ty=QQ|NN
and
(*Les modes *)
   Mo=GG|HH|VV|AA
and
   (*L'information sur le cut*)
      Cf=CU|NC
and
(*Les sequences abstraites atomiques*)
      Aass=INF
      |A_ZERO
      |UN_S
      |UN_P
      |DEUX
      |DEUX_P
and
(*Les sequences abstraites atomiques avec cut information*)
      Aass_Cut=A_C of (Aass*Cf)
and
```

```
(*Les composants modes*)
        Modes=M of (int*Mo)list
and
(*Les composants types*)
        Types=T of (int*Ty)list
and
(*Les substitutions concretes*)
        Sub_Con=S_Cc of (int*Terme)list
and
(*Les composants liaisons*)
        Liaisons=L of ((int*int)list*(int*(int list))list)
and
(*Les substitutions abstraites*)
        Sub_Abst=S_Ab of (Modes*Types*Liaisons)
        |SA_VIDE
and
(*Les sequences abstraites*)
        Seq_Abst=Se_Ab of (Sub_Abst*(Aass list))
        |SEA_VIDE
and
(*Les sequences abstraites avec l'information cut*)
        Seq_Abst_Cut=Sec_Ab of Sub_Abst*((Aass*Cf)list)
        |SECA_VIDE;;
type
        Programme=PG of (Procedure list)
and
        Procedure=PR of string*(Clause list)
and
        Clause=CL of (Tete*Corps)
and
        Tete=TE of string*(int list)
and
        Corps=CO of (Atome list)
        |VIDE
and
        Atome=AP of string*(int list)
        |UN of (Terme*Terme)
        |FU of (Terme*Terme)|CUT;;
```

A.2 Quelques fonctions de base

```
(*Cette fonction associe un entier a chaque type d'atomes*)
   let type_atome=function
       (AP x)- >1
       | (FU ((VAR x),ZERO))- >2
       | (FU ((VAR x),s(ZERO)))- >3
       | (FU (ZERO,(VAR x)))- >4
       | (FU (s(ZERO),(VAR x)))- >5
       | (FU (s(VAR x),(VAR y)))- >if x=y then failwith"fail_occur" else 6
       | (FU ((VAR x),s(VAR y)))- >if x=y then failwith"fail_occur" else 7
       | (UN ((VAR x),(VAR y)))- >if x=y then failwith"expression_inutile" else 8
       | CUT- >9
       |_- >failwith"Erreur de syntaxe";;
(*A partir d'un programme, cette fonction contenu d'une procedure de nom donne *)
   let rec rech_pro proc =function
       []- >[]
       |(PR(x,y))::l- >if proc=x then y else rech_pro proc l
       | _- >failwith "Cas non prevu" ;;
(*Cette fonction renvoie le nom d'une clause recue*)
       let name_clause =function

       (CL((TE(c,d)),b))- >c
       |(CL((TE(c,d)),b))- >c
       |_- >failwith "Cas non prevu" ;;
(*Cette fonction renvoie le nom de la procedure d'un atome d'appel*)
   let name_proc_appel=function
       (AP (x,y))- >x
       | _- >failwith "Cas non prevu" ;;
(*Cette fonction retourne le premier composant d'un triplet*)
   let proj1 (a,b,c)=a ;;
(*Cette fonction retourne le deuxieme composant d'un triplet*)
   let proj2 (a,b,c)=b ;;
(*Cette fonction retourne le troisieme composant d'un triplet*)
   let proj3 (a,b,c)=c ;;
(*Cette fonction retourne le couple definissant un atome d'appel*)
   let couple (AP (x,y))=(x,y) ;;
(*Cette fonction retourne le corps d'une clause*)
   let corp=function
       (CL(x,(CO y)))- >y
       |_- >failwith "Cas non prevu" ;;
```

```
(*Cette fonction retourne le triplet definissant une substitution abstraite*)
   let triplet (S_Ab x)=x;;
(* Ces fonctions retournent verifient , selon les cas, l'existence d'un element dans une liste *)
   let rec existe_elt a =function
       []- >false
       |c::k- >if (a=c) then true else existe_elt a k;;
   let rec existe_cple a =function
       []- >false
       |(c,d)::k- >if (a=c) then true else existe_cple a k;;
   let rec existe_cple_s (a,b)=function
       []- >false
       |(c,d)::k- >if ((a=c) & (b=d)) then true else existe_cple_s (a,b) k;;
   let rec existe_cple_c (a,b)=function
       []- >false
       |((c,d),l)::k- >if ((a=c) & (b=d)) then true else existe_cple_c (a,b) k;;
(*A partir d'une liste, ces fonctions retourne l'extremite du couple dont l'origine est donnee*)
   let rec chercher_cple (b,aa)=function
       []- >[]
       |((c,d),l)::k- >if ((aa=d) & (b=c)) then l else chercher_cple (b,aa) k;;
   let rec chercher_elt aa=function
       []- >failwith"L'indice n'est pas repris dans la liste"
       |(d,l)::k- >if (aa=d) then l else chercher_elt aa k;;
(*Ces fonctions nettoient les listes de maniere a ce que chaque element soit
repris une et une seule fois*)
   let rec eliminer_elt =function
       []- >[]
       |e::l- >if (existe_elt e l) then eliminer_elt l else e::(eliminer_elt l);;
   let rec eliminer_cple =function
       []- >[]|
       (e,f)::l- >if (existe_cple_s (e,f) l) then eliminer_cple l
       else (e,f)::(eliminer_cple l);;
(*Ces fonctions trier, selon les cas, une liste donnee.*)
   let rec trier_elt=
       let rec insert_elt q= function
          [] - > [q]
       | p::l - > if (q¡p) then q::p::l else p::(insert_elt q l)
   in function
       [] - > []
       |q::l - > insert_elt q (trier_elt l);;
```

```
let rec trier_cple =
    let rec insert_cple (q,ss)= function
    [] -> [(q,ss)]
    |(p,t)::l -> if (q¡p) then (q,ss)::(p,t)::l
    else (p,t)::(insert_cple (q,ss) l) in
    function
    []-> []
    | (q,ss)::l -> insert_cple (q,ss) (trier_cple l);;
(*Cette fonction change la valeur correspondant a un indice donne*)
let rec changer_val i val=function
    []->[]
    |(a,b)::l-> if a=i then (a,val)::l else (a,b)::(changer_val i val l);;
let changer_mo (((M m),(T t),(L (l1,l2)))) i (a,b)=
    if ((b=VV)&
    ((existe_cple_s (i,a) l1) or (existe_cple_s (a,i) l1)))
    then (a,GG)
    else (a,b);;
(*Cette fonction implemente l'operation changer_mode_GG*)
let changer_mode (((M m),(T t),(L (l1,l2)))) i l=
    map (changer_mo (((M m),(T t),(L (l1,l2)))) i )l;;
let changer_mo1 (((M m),(T t),(L (l1,l2)))) i (a,b)=
    if ((b=VV)&
    ((existe_cple_s (i,a) l1) or (existe_cple_s (a,i) l1)))
    then (a,AA)
    else (a,b);;
(*Cette fonction implemente l'operation changer_mode_AA*)
let changer_mode1 (((M m),(T t),(L (l1,l2)))) i l=
    map(changer_mo1 (((M m),(T t),(L (l1,l2)))) i )l;;
let changer_modee x l i=changer_mode1 x i l;;
let changer_mode1_liste x l1 g=map (changer_modee x l1) g;;
let changer_ty (((M m),(T t),(L (l1,l2)))) i (a,b)=
    if ((b=QQ)&
    ((existe_cple_s (i,a) l1) or (existe_cple_s (a,i) l1)))
    then (a,NN)
    else (a,b);;
(*Cette fonction implemente l'operation changer_type*)
let changer_type ( ((M m),(T t),(L (l1,l2)))) i l=
    map(changer_ty ( ((M m),(T t),(L (l1,l2)))) i )l;;
(* recherche du domaine d'indices*)
let rec dom=function
    []-> []
    |(a,b)::l-> a::dom l;;
```

```
(* Cette fonction retourne TRUE ssi une sequence avec l'information sur le cut est infinie ou
   si elle est relative a un cut*)
   let cut_or_nt=
       let c_or_n (x,cf)=(cf=CU) or (x=inF) or (x=UN_P) or (x=DEUX_P) in
       let rec cu_or_n =function
       []- >false
       | x::reste- >if (c_or_n x) then true else cu_or_n reste in
   function
       SECA_VIDE- >false
       |(Sec_Ab(y,LC))- >cu_or_n LC;;
   let cut_or_nts x=mem true (map cut_or_nt x);;

(*Cette fonction sera utilisee pour renommer une substitution*)
       let rec new_num1 n =function
       []- >[]
       |a::l- >(a,10000+n)::(new_num1 (n+1) l);;
   let rec ajout_nocut =function
       []- >[]
       |a::k- >(a,NC)::ajout_nocut k;;
   let rendre_sec (Se_Ab(x,y))=(Sec_Ab(x,ajout_nocut y));;
```

Annexe B

Les opérations Abstraites

B.1 L'entrée d'une clause

SIGNATURE:
EXTC : Clause $\mapsto$ Sub_Abst $\mapsto$ Seq_Abst_Cut = $< fun >$

```
let EXTC (CL ((TE (te,liste)),(CO co))) (S_Ab ((M m),(T t),(L (l1,l2))))=

    let var_terme =function
      (VAR x) - > [x]
      | s(VAR x)- >[x]
      | _- >[] in
    let rec liste_var_corps=function
      [] - > [] |
      (AP(x,y))::reste - > eliminer_elt(y@(liste_var_corps reste)) |
      (UN(x,y))::reste - > eliminer_elt((var_terme x)@(var_terme x)@
          (liste_var_corps reste))|
      (FU(x,y))::reste - > eliminer_elt((var_terme x)@(var_terme x)@
          (liste_var_corps reste))|
      CUT::reste- >(liste_var_corps reste) in
    let rec ajout_qlcq liste val=function
      []- >[] |
      a::l- >(a,val)::(ajout_qlcq liste val l) in
    let rec ajout_vide l =function
      []- >[]|
      a::k- >(a,[])::(ajout_vide l k) in
    let difference =subtract (liste_var_corps co) liste in
    let nouveau_mode=trier_cple(m@(ajout_qlcq m HH difference)) in
```

```
        let nouveau_type=trier_cple(t@(ajout_qlcq m QQ difference)) in
        let nouveau_liaison=(l1,trier_cple((ajout_vide l2 difference)@l2)) in
(Sec_Ab (((S_Ab((M nouveau_mode),(T nouveau_type),(L nouveau_liaison))),[(UN_S,NC)])));;
```

B.2 La sortie d'une clause

SIGNATURES:
RESTRC: Clause $\mapsto$ Seq_Abst_Cut $\mapsto$ Seq_Abst_Cut = $< fun >$
RESTRCS: Clause $\mapsto$ Seq_Abst_Cut list $\mapsto$ Seq_Abst_Cut list = $< fun >$

```
let RESTRC (CL ((TE (te,liste)),(CO co)))(Sec_Ab((S_Ab ((M m),(T t),(L (l1,l2)))),LC))=

    let rec effacer l=function
       []- >[]|
       (a,b)::k- >if (existe_elt a l) then (a,b)::(effacer l k)
           else effacer l k in
    let rec effacer_l1 l=function
       []- >[]|
       (a,b)::k- >if ((existe_elt a l)&(existe_elt b l)) then
           (a,b)::(effacer_l1 l k)else effacer_l1 l k in
    let rec effacer_l2 l=function
       []- >[]|
       (a,b)::k- >if (existe_elt a l) then (a, intersect l b)::(effacer_l2 l k)
           else effacer_l2 l k in
    let nouveau_mode=effacer liste m in
    let nouveau_type=effacer liste t in
    let nouveau_l1=effacer_l1 liste l1 in
       (Sec_Ab((S_Ab((M nouveau_mode),(T nouveau_type),(L( l1, l2)))),LC));;
let RESTRCS x l=map (RESTRC x) l;;
```

B.3 La préparation à un appel

SIGNATURE:
RESTRG : Atome $\mapsto$ Sub_Abst $\mapsto$ Sub_Abst = $< fun >$

```
let RESTRG= fun
(AP(nom_proc, liste_var)) (S_Ab((M m),(T t),(L (l1,l2))))- >
   let rec new_num n =function
       []- >[]|
       a::l- >(n,10000+a)::(new_num (n+1) l) in
   let rec che (a,b) =function
       []- >failwith"il y a un probleme"|
       (c,d)::k- >if (b-10000)=c then (a,d) else che (a,b) k in
   let rec cherche l =function
       []- >[]|
       (a,b)::k- >(che (a,b) l)::(cherche l k) in
   let rec f1 a b c=if (a=c) then b else c in
   let rec f2 (a,b)=function
       []- >[]|
       (c,d)::k- >(f1 a b c,f1 a b d)::(f2 (a,b) k) in
   let rec f22 (a,b)=function
       []- >[]|
       c::k- >(f1 a b c)::(f22 (a,b) k) in
   let rec f3 l=function
       []- >l|
       (a,b)::k- >f3 (f2 (a,b) l) k in
   let rec f4=function
       []- >[]|
       (a,b)::k- >(a-10000,b-10000)::(f4 k) in
   let rec f44=function
       []- >[]|
       a::k- >trier_elt((a-10000)::(f44 k)) in
   let rec f5 (a,b)=function
       []- >[]|
       (c,d)::k- > (f1 a b c, f22 (a,b) d )::(f5 (a,b) k) in
   let rec f33 l=function
       []- >l|
       (a,b)::k- >f33 (f5 (a,b) l) k in
   let rec eliminer_unitiles_cple k =function
       []- >[]|
       (a,b)::l- >if (existe_elt a k)&(existe_elt b k)
          then (a,b)::eliminer_unitiles_cple k l
          else eliminer_unitiles_cple k l in
```

```
let rec f6 =function
    []- >[]|
    (c,d)::k- >(c-10000, f44 d )::(f6 k) in
let rec eliminer_unitiles_elt k =function
    []- >[]|
    a::l- >if (existe_elt a k) then a::eliminer_unitiles_elt k l
    else eliminer_unitiles_elt k l in
let rec eliminer_unitiles_c k =function
    []- >[]|
    (a,b)::l- >if (existe_elt a k)
        then (a, eliminer_unitiles_elt k b)::eliminer_unitiles_c k l
        else eliminer_unitiles_c k l in
let n=list_length liste_var in
let n1=list_length m in
let c=new_num 1 liste_var in
let cc=new_num1 1 liste_var in
let nouveau_mode=trier_cple(cherche m c) in
let nouveau_type=trier_cple(cherche t c) in
let nouveau_l1=if n=n1 then f4(f3 l1 c)
    else if n<n1 then
        let l11= eliminer_unitiles_cple liste_var l1 in
        let l111=f3 l11 cc in
        else failwith"Store trop petit pour repondre a cet appel" in
let nouveau_l2=if n=n1 then trier_cple (f6(f33 l2 c))
    else if n<n1 then
    let l22=eliminer_unitiles_c liste_var l2 in
    let l222=f33 l22 cc in
    trier_cple(f6 l222)
    else failwith"Il y a un Probleme" in
(S_Ab((M nouveau_mode),(T nouveau_type),(L (nouveau_l1,nouveau_l2))))|
x y- >y;;
```

B.4 Les deux unifications

SIGNATURES:
UNIF_FUNC : Atome $\mapsto$ Sub_Abst $\mapsto$ Seq_Abst = $< fun >$
UNIF_VAR : Atome $\mapsto$ Sub_Abst $\mapsto$ Seq_Abst = $< fun >$

```
let rec maj_mode_type f o va=fun
    []- >[]|
    (a,b)::l- >if a=o then (a,va)::l
        else (f (a,b))::(maj_mode_type f o va l);;
let rec placer_premier a =fun
    []- >[]|
    (c,d)::l- >if c=a then (c,d)::l else (placer_premier a l)@[(c,d)];;
let rec placer_premiers i j l=
    (hd (placer_premier i l))::(placer_premier j (tl (placer_premier i l)));;
    let couper2=fun
    (a,b)::(c,d)::l- >(b,d,l)|
    _- >failwith "unification impossible";;
let couper1=fun
    (a,b)::l- >(a,b,l)|
    _- >failwith "unification impossible";;
let rec ch[]2 i j=fun
    (a,b)::k- >if a=i then (a,eliminer_elt(j::b))::ch_l2 i j k else
        if a=j then (a,eliminer_elt(i::b))::ch_l2 i j k else
        (a,b)::ch_l2 i j k;;
let rec unif_va i j ((M m),(T t),(L(l1,l2)))=
    let temp=((M m),(T t),(L(l1,l2))) in
    fun

        GG GG NN NN- >(Se_Ab(S_Ab((M(trier_cple((i,GG)::(j,GG)::m))),

            (T(trier_cple((i,NN)::(j,NN)::t))),(L(l1,ch_l2 i j l2))),[A_ZERO;UN_S]))
        | GG HH NN QQ- >(Se_Ab(S_Ab((M(trier_cple((i,GG)::(j,GG)::m))),

            (T(trier_cple((i,NN)::(j,NN)::t))),(L(l1,ch_l2 i j l2))),[UN_S]))
        |GG VV NN QQ- >(Se_Ab(S_Ab((M(trier_cple((i,GG)::(j,GG)::(changer_mode temp j m)))),

            (T(trier_cple((i,NN)::(j,NN)::(changer_type temp j t)))),(L(l1,ch_l2 i j l2))),[UN_S]))|

        |GG AA NN QQ- >(Se_Ab(S_Ab((M(trier_cple((i,GG)::(j,GG)::(changer_mode temp j m)))),

            (T(trier_cple((i,NN)::(j,NN)::(changer_type temp j t)))),(L(l1,ch_l2 i j l2))),

            [A_ZERO;UN_S]))|
```

```
|HH HH QQ QQ- >(Se_Ab(S_Ab((M(trier_cple((i,VV)::(j,VV)::m))),
   (T(trier_cple((i,QQ)::(j,QQ)::t))),(L((eliminer_cple(l1@[(i,j);(j,i)])),ch_l2 i j l2))),
   [UN_S]))
|HH VV QQ QQ- >(Se_Ab(S_Ab((M(trier_cple((i,VV)::(j,VV)::m))),
   (T(trier_cple((i,QQ)::(j,QQ)::t))),(L((eliminer_cple(l1@[(i,j);(j,i)])),ch_l2 i j l2))),
   [UN_S]))
| HH AA QQ QQ- >(Se_Ab(S_Ab((M(trier_cple((i,AA)::(j,AA)::m))),
   (T(trier_cple((i,QQ)::(j,QQ)::t))),(L((eliminer_cple(l1@[(i,j);(j,i)])),ch_l2 i j l2))),
   [UN_S]))
|VV VV QQ QQ- >(Se_Ab(S_Ab((M(trier_cple((i,VV)::(j,VV)::m))),
   (T(trier_cple((i,QQ)::(j,QQ)::t))),(L((eliminer_cple(l1@[(i,j);(j,i)])),ch_l2 i j l2))),
   [UN_S]))
| VV AA QQ QQ- >let temp1=((M (changer_val j AA m)),(T t),(L(l1,ch_l2 i j l2))) in
   (Se_Ab(S_Ab((M(trier_cple((i,AA)::(j,AA)::
   (changer_model temp1 i (changer_model temp1 j m))))),
   (T(trier_cple((i,QQ)::(j,QQ)::t))),(L((eliminer_cple(l1@[(i,j);(j,i)])),ch_l2 i j l2))),
   [A_ZERO;UN_S]))
|AA AA QQ QQ- >(Se_Ab(S_Ab((M(trier_cple((i,AA)::(j,AA)::(changer_model temp i
   (changer_model temp j m))))),
   (T(trier_cple((i,QQ)::(j,QQ)::t))),(L((eliminer_cple(l1@[(i,j);(j,i)])),ch_l2 i j l2))),
   [A_ZERO;UN_S]))
x y w z- > unif_va j i temp y x z w;;

let UNIF_VAR (UN((VAR i),(VAR j))) (S_Ab((M m),(T t),(L(l1,l2))))=

   let new_m=placer_premiers i j m in
   let new_t=placer_premiers i j t in
   let (coord1_i,coord1_j,m1)=couper2 new_m in
   let (coord2_i,coord2_j,t1)=couper2 new_t in
   unif_va i j ((M m1),(T t1),(L(l1,l2))) coord1_i coord1_j coord2_i coord2_j;;
```

```
    let unif_f_cas2 (FU(VAR x,y)) (S_Ab((M m),(T t),(L(l1,l2))))=
    let temp=((M m),(T t),(L(l1,l2))) in
    let (xx1,xxx2,new_m)=couper1(placer_premier x m) in
    let (xx2,xxx1,new_t)=couper1(placer_premier x t) in
    let unif1 =
    fun
    GG NN- >(Se_Ab(S_Ab((M(trier_cple((xx1,GG)::new_m))),

        (T(trier_cple((xx2,NN)::new_t))),(L(l1,l2))),[A_ZERO;UN_S]))

    | HH QQ- >(Se_Ab(S_Ab((M(trier_cple((xx1,GG)::new_m))),

        (T(trier_cple((xx2,NN)::new_t))),(L(l1,l2))),[UN_S]))

    | VV QQ- >(Se_Ab(S_Ab((M(trier_cple((xx1,GG)::(changer_mode temp xx1 new_m)))),

        (T(trier_cple((xx2,NN)::(changer_type temp xx2 new_t)))),(L(l1,l2))),[UN_S]))
    | AA QQ- >(Se_Ab(S_Ab((M(trier_cple((xx1,GG)::(changer_mode temp xx1 new_m)))),

        (T(trier_cple((xx2,NN)::(changer_type temp xx2 new_t)))),(L(l1,l2))),

        [A_ZERO;UN_S])) in

    unif1 xxx2 xxx1 ;;
¿let unif_f (S_Ab((M m),(T t),(L(l1,l2))))=
    let sub=(S_Ab((M m),(T t),(L(l1,l2)))) in
    fun (FU(VAR x,y))- >unif_f_cas2 (FU(VAR x,y)) sub|
        (FU(y,VAR x))- >unif_f_cas2 (FU(VAR x,y)) sub;;
    let rec maj_liste2 (x,y)=fun
        []- >[]|
        (a,b)::l- >if a=x then (a,y::b)::l else
        (a,b)::(maj_liste2 (x,y) l) ;;
    let unif_f1_cas6 (FU(s(VAR x),(VAR y)))(S_Ab((M m),(T t),(L(l1,l2))))=
    let temp=((M m),(T t),(L(l1,l2))) in
    let new_m=placer_premiers x y m in
    let new_t=placer_premiers x y t in
    let (coord1_x,coord1_y,m1)=couper2 new_m in
    let (coord2_x,coord2_y,t1)=couper2 new_t in
    let new_l2=maj_liste2 (y,x) l2 in
    let rec unif1 i j=
fun
    GG GG NN NN- >(Se_Ab(S_Ab((M(trier_cple((i,GG)::(j,GG)::m1))),

        (T(trier_cple((i,NN)::(j,NN)::t1))),(L(l1,new_l2))),[A_ZERO;UN_S]))
```

```
|GG HH NN QQ– >(Se_Ab(S_Ab((M(trier_cple((i,GG)::(j,GG)::m1))),

    (T(trier_cple((i,NN)::(j,NN)::t1))),(L(l1,new_l2))),[UN_S]))
|GG VV NN QQ– >(Se_Ab(S_Ab((M(trier_cple((i,GG)::(j,GG)::(changer_mode temp j m1)))),

    (T(trier_cple((i,NN)::(j,NN)::(changer_type temp j t1)))),(L(l1,new_l2))),

    [UN_S]))
|GG AA NN QQ– >(Se_Ab(S_Ab((M(trier_cple((i,GG)::(j,GG)::(changer_mode temp j m1)))),

       (T(trier_cple((i,NN)::(j,NN)::(changer_type temp j t1)))),(L(l1,new_l2))),

    [A_ZERO;UN_S]))
|HH HH QQ QQ– >(Se_Ab(S_Ab((M(trier_cple((i,VV)::(j,VV)::m1))),

    (T(trier_cple((i,QQ)::(j,QQ)::t1))),(L((eliminer_cple(l1@[(i,j);(j,i)])),new_l2))),

    [UN_S]))
|HH VV QQ QQ– >(Se_Ab(S_Ab((M(trier_cple((i,VV)::(j,VV)::m1))),
       (T(trier_cple((i,QQ)::(j,QQ)::t1))),(L((eliminer_cple(l1@[(i,j);(j,i)])),new_l2))),

    [UN_S]))
|HH AA QQ QQ– >(Se_Ab(S_Ab((M(trier_cple((i,AA)::(j,AA)::m1))),
       (T(trier_cple((i,QQ)::(j,QQ)::t1))),(L((eliminer_cple(l1@[(i,j);(j,i)])),new_l2))),

    [UN_S]))
|VV VV QQ QQ– >(Se_Ab(S_Ab((M(trier_cple((i,VV)::(j,VV)::m1))),

    (T(trier_cple((i,QQ)::(j,QQ)::t1))),(L((eliminer_cple(l1@[(i,j);(j,i)])),new_l2))),

    [UN_S]))
|VV AA QQ QQ– >let temp1=((M (changer_val j AA m)),(T t),(L(l1,l2))) in
    (Se_Ab(S_Ab((M(trier_cple((i,AA)::(j,AA)::(changer_model temp1 i

    (changer_model temp1 j m1))))),

       (T(trier_cple((i,QQ)::(j,QQ)::t1))),(L((eliminer_cple(l1@[(i,j);(j,i)])),new_l2))),

    [A_ZERO;UN_S]))
| AA AA QQ QQ– >(Se_Ab(S_Ab((M(trier_cple((i,AA)::(j,AA)::

    (changer_model temp i (changer_model temp j m1))))),

       (T(trier_cple((i,QQ)::(j,QQ)::t1))),(L((eliminer_cple(l1@[(i,j);(j,i)])),new_l2))),
```

```
    [A_ZERO;UN_S]))|
x y w z- > unif1 j i y x z w in
unif1 x y coord1_x coord1_y coord2_x coord2_y ;;

let unif_f1 (S_Ab((M m),(T t),(L(l1,l2))))= fun

(FU(s(VAR x),(VAR y)))- >

    unif_f1_cas6 (FU(s(VAR x),(VAR y)))(S_Ab((M m),(T t),(L(l1,l2))))|
    (FU((VAR x),s(VAR y)))- >

    unif_f1_cas6 (FU(s(VAR y),(VAR x)))(S_Ab((M m),(T t),(L(l1,l2))));;

let UNIF_FUNC (FU(x,y)) (S_Ab((M m),(T t),(L(l1,l2))))=

    let n =type_atome (FU(x,y)) in
    if (n>1)&(n<6) then unif_f (S_Ab((M m),(T t),(L(l1,l2)))) (FU(x,y)) else

    if (n=6)or(n=7) then unif_f1 (S_Ab((M m),(T t),(L(l1,l2)))) (FU(x,y)) else
failwith"Cas non prevu";;
```

B.5 L'interprétation de *cut* et les deux extractions

SIGNATURES:
AI_CUT : Seq_Abst_Cut $\mapsto$ Seq_Abst_Cut = $< fun >$
SEQ : Seq_Abst_Cut $\mapsto$ Seq_Abst = $< fun >$
SEQS : Seq_Abst_Cut list $\mapsto$ Seq_Abst list = $< fun >$
SUBST : Seq_Abst_Cut $\mapsto$ Sub_Abst = $< fun >$

```
let AI_CUT (Sec_Ab(x, LC))=
    let ai_1 = function
    (INF,x)- >(INF,x)|
    (A_ZERO,x)- >(A_ZERO,x)|
    (y,x)- >(UN_S,CU) in
    let ai_liste l=eliminer_cple (map ai_1 l) in
(Sec_Ab(x, ai_liste LC));;

    let se_1(x,y)=x in
    let se_list l=map se_1 l in
(Se_Ab(x, se_list LC));;

let SEQS x= map SEQ x;;

let SUBST (Sec_Ab(x, LC))=x;;
```

B.6 L'extension du résultat d'un appel

SIGNATURES:
EXTG : Atome $\mapsto$ Seq_Abst_Cut $\mapsto$ Seq_Abst $\mapsto$ Seq_Abst_Cut = $< fun >$
EXTGS : Atome $\mapsto$ Seq_Abst_Cut $\mapsto$ Seq_Abst list $\mapsto$ Seq_Abst_Cut list = $< fun >$

```
let inter_type=fun
    QQ QQ- >QQ|
    - -- >NN;;

let rec inter_liste =fun
    [][]- >[]|
    ((a,b)::l)((x,y)::g)- >trier_cple((x,inter_type b y)::(inter_liste l g))|
    - -- >failwith"Cas non prevu";;

let rec changer_model_liste (((M m1),(T t1),(L (l11,l21)))) liste_a_traiter=
    let triple_sub=(((M m1),(T t1),(L (l11,l21)))) in
    function
        []- >liste_a_traiter|
        (i,x)::reste- >
        if x=AA then
        changer_model_liste triple_sub (changer_model triple_sub i liste_a_traiter) reste
        else changer_model_liste triple_sub liste_a_traiter reste;;

let rec produit_cart b = function
    []- >[]|
    a::l- >(a,b):: (produit_cart b l);;

let rec inc = function
    []- >[]|
    A_ZERO:: reste- >INF::(inc reste)|
    UN_S::reste- >UN_P::(inc reste)|
    DEUX::reste- >DEUX_P::(inc reste)|
    x::reste- >x::(inc reste);;

let rec cas_2 = function
    []- >[]|
    UN_S::reste- >DEUX::(cas_2 reste)|
    x::reste- >x::(cas_2 reste);;
```

```
let rec cas_2_P= function
    []- >[]|
    A_ZERO::reste- >INF::(cas_2_P reste)|
    UN_S::reste- >DEUX_P ::(cas_2_P reste)|
    DEUX::reste- >DEUX_P::(cas_2_P reste)|
x::reste- >x::(cas_2_P reste);;

let rec extg_aux=fun
    (INF,cf) _- >[(INF,cf)]|
    (A_ZERO,cf) _- >[(A_ZERO,cf)]|
    (UN_S,cf) B- >produit_cart cf B|
    (UN_P,cf) B- >produit_cart cf (eliminer_elt(inc B))|
    (DEUX,cf) B- >produit_cart cf (eliminer_elt(cas_2 B))|
    (DEUX_P,cf) B- >produit_cart cf (eliminer_elt(cas_2_P B));;

let rec extg B = function
    []- >[]|x::reste- >eliminer_cple((extg_aux x B)@(extg B reste));;

let rec sa_valeur var= function
    []- >failwith"la variable n'est pas reprise dans le store"|
      (c,d)::reste- >if var=c then d else sa_valeur var reste;;

let modifier_mo_elt (a,b) m (c,d) =
    let e=if (existe_cple a m) then (sa_valeur a m) else HH in
      if (((b-10000)=c) &(e<>HH)&(d=HH)) then (a,VV)
      else (a,d) ;;

let modifier_ty_elt (a,b) t (c,d) =
    let e=if (existe_cple a t) then (sa_valeur a t) else QQ in
      if ((b-10000)=c) then (a, inter_type e d)
      else (a,d) ;;

    let modifier_mo_liste (a,b) m m1=map (modifier_mo_elt (a,b) m) m1 ;;

    let modifier_ty_liste (a,b) t t1=map (modifier_ty_elt (a,b) t) t1 ;;


    let rec modifier_mo_tous m m1= function
      []- >m|
      (a,b)::num1- >modifier_mo_tous (modifier_mo_liste (a,b) m m1) m1 num1 ;;




    let rec modifier_ty_tous t t1= function
```

```
   []- >t|
   (a,b)::num1- >modifier_ty_tous (modifier_ty_liste (a,b) t t1) t1 num1 ;;
let rec restant liste1=
let doma=dom liste1 in
   function

   []- >[]|
   (a,b)::liste2- >if (existe_elt a doma) then (restant liste1 liste2) else
   (a,b)::(restant liste1 liste2) ;;

let rec changer_mod l1 i = function
   []- >[]|  (a,b)::reste- >if ( (existe_cple_s (i,a) l1) or (existe_cple_s (a,i) l1))
   then (a,GG)::(changer_mod l1 i reste)
   else (a,b)::(changer_mod l1 i reste) ;;

let rec changer_typ l1 i = function
   []- >[]|
   (a,b)::reste- >if ((existe_cple_s (i,a) l1) or (existe_cple_s (a,i) l1))
   then (a,NN)::(changer_typ l1 i reste)
   else (a,b)::(changer_typ l1 i reste) ;;
let rec appliquer_l1_m l1 m= function
   []- >m|
   (a,b)::reste- >if b<>GG then (appliquer_l1_m l1 m reste) else
   appliquer_l1_m l1 (changer_mod l1 a m) reste;;

let rec appliquer_l1_t l1 t= function
   []- >t|

   (a,b)::reste- >if b¡¿NN then (appliquer_l1_t l1 t reste) else
   appliquer_l1_t l1 (changer_typ l1 a t) reste;;
let EXTG =fun
   x y SEA_VIDE- > y|
   x (Sec_Ab(sub,lc)) (Se_Ab(SA_VIDE,l))- >(Sec_Ab(sub,extg l lc))|
   x (Sec_Ab(SA_VIDE,lc)) (Se_Ab(sub,l))- >(Sec_Ab(SA_VIDE,[(A_ZERO,NC)]))|
   (Se_Ab((S_Ab((M m1),(T t1),(L (l11,l21)))),L1))- >
      let cc=new_num1 1 (intersect liste_var (dom m)) in

      let nouveau_m =modifier_mo_tous m m1 cc in

   let nouveau_t =modifier_ty_tous t t1 cc in

   let new_m=trier_cple(eliminer_cple((nouveau_m)@(restant nouveau_m m))) in

   let new_t=trier_cple(eliminer_cple((nouveau_t)@(restant nouveau_t t))) in
```

```
            let temp=(((M new_m),(T new_t),(L (l1,l2)))) in

            let neww_m=changer_model_liste temp new_m new_m in

            let newww_m=appliquer_l1_m l1 neww_m neww_m in
            let newww_t=appliquer_l1_t l1 new_t new_t in

            let new_LC=extg L1 LC in

    (Sec_Ab((S_Ab((M newww_m),(T newww_t),(L (l1,l2)))),new_LC))|
_ (Sec_Ab((S_Ab((M m),(T t),(L (l1,l2)))),LC))
(Se_Ab((S_Ab((M m1),(T t1),(L (l11,l21)))),L1))- >
            let new_LC=extg L1 LC in
                (Sec_Ab((S_Ab((M m1),(T t1),(L (l11,l21)))),new_LC))

    let EXTGS x y = function
        []- >[y]|
        l- >eliminer_elt(map (EXTG x y) l);;
```

B.7 La concaténation

SIGNATURES:
UNION : Sub_Abst $\mapsto$ Sub_Abst $\mapsto$ Sub_Abst = $< fun >$
NOT_EXCLUSIVE : Sub_Abst $\mapsto$ Sub_Abst $\mapsto$ Sub_Abst $\mapsto$ bool = $< fun >$
CONCaux : Seq_Abst_Cut $\mapsto$ Seq_Abst $\mapsto$ Seq_Abst = $< fun >$
CONC : Sub_Abst $\mapsto$ Seq_Abst_Cut $\mapsto$ Seq_Abst $\mapsto$ Seq_Abst list = $< fun >$
CONCS : Sub_Abst $\mapsto$ Seq_Abst_Cut list $\mapsto$ Seq_Abst list $\mapsto$ Seq_Abst list =$< fun >$

```
let rec max_m=fun
    GG GG− >GG|
    GG HH− >AA|
    GG VV− >AA|
    GG AA− >AA|
    HH HH− >HH|
    HH VV− >VV|
    HH AA− >AA|
    VV VV− >VV|
    VV AA− >AA|
    AA AA− >AA|
    x y− >max_m y x;;

let rec max_t=fun
    NN NN− >NN|
    _ _− >QQ;;

let rec max_mode_type f=fun
    [][]− >[]|
    ((a,b)::m_t)((c,d)::m1_t1)− >if (a=c) then (a,f b d)::max_mode_type f m_t m1_t1 else
       failwith"Il y a un probleme d'indice"|
    _ _− >failwith"Impossible de calculer max_mode";;
let UNION=fun
    SA_VIDE x− >x|
    x SA_VIDE− >x|
    (S_Ab((M m),(T t),(L (l1,l2)))) (S_Ab((M m1),(T t1),(L (l11,l21))))− >
       (S_Ab((M (max_mode_type max_m m m1)),(T (max_mode_type max_t t t1)),
       (L (eliminer_cple(l1@l11),eliminer_cple(l2@l21)))));;
let rec UNION_liste= function
    []− >SA_VIDE|
    a::[]− >a|
    a::b::reste− > let x=(UNION a b) in (UNION x (UNION_liste reste));;
let rec ground= function
    []− >[]|
```

```
    (a,b)::reste- >if (b=GG) then trier_elt(a::(ground reste))
       else (ground reste) ;;
let rec f_zero= function
    []- >[]|
    (a,b)::reste- >if (b=[]) then trier_elt(a::(f_zero reste)) else f_zero reste;;
let rec not_exlus l1 l2= function
    []- >true|
    a::l- >if ((existe_elt a (f_zero l1))=(existe_elt a (f_zero l2)))
       then (not_exlus l1 l2 l) else false;;

let NOT_EXCLUSIVE= fun
    (S_Ab((M m),(T t),(L (l1,l2))))
    (S_Ab((M m1),(T t1),(L (l11,l21))))
    (S_Ab((M m2),(T t2),(L (l12,l22))))- >
let l=ground m in
    not_exlus l21 l22 l |
    _ (SA_VIDE)_- >true|
    _ _(SA_VIDE)- >true|
    (SA_VIDE) _ _- >true;;
let rec add =fun
    INF _- >INF|
    UN_P _- >UN_P|
    DEUX_P _- >DEUX_P|
    A_ZERO x- >x|
    UN_S INF- >UN_P|
    UN_S A_ZERO- >UN_S|
    UN_S UN_S- >DEUX|
    UN_S UN_P- >DEUX_P|
    UN_S DEUX- >DEUX_P|
    UN_S DEUX_P- >DEUX_P|
    DEUX A_ZERO- >DEUX|
    DEUX x- >DEUX_P|
    x y- >add y x;;

let rec add_l = function
    []- >[]|
    [x]- >[x]|
    x::y::l- >add_l ((add x y)::l);;



let rec max_Aa=fun
    INF x- >x|
    A_ZERO x- >if x=INF then A_ZERO else x |
    UN_S x- > if x=INF then UN_S else x|
```

```
    UN_P x- > if (x=INF) or(x=UN_S) then UN_P else x|
    DEUX x- > if (x=DEUX_P) then x else DEUX|
    DEUX_P x- >DEUX_P|
    x y- >max_Aa y x;;
let rec addition_l= function
    []- >[]|
    [a]- >[a]|
    a::b::l- >addition_l ((max_Aa a b)::l);;


let CONCaux (Sec_Ab(sub,[(b1,cf1)]))(Se_Ab(sub1,[b2]))=
    (Se_Ab(UNION sub sub1, [addition (b1,cf1) b2]));;

let rec CONCaux_elt_liste sub (Sec_Ab(sub1,[(b1,cf1)]))=
    []- >[]|
    (Se_Ab(sub2,[b2]))::reste- >if (NOT_EXCLUSIVE sub sub1 sub2) then
       (CONCaux (Sec_Ab(sub1,[(b1,cf1)]))(Se_Ab(sub2,[b2])))::
       (CONCaux_elt_liste sub (Sec_Ab(sub1,[(b1,cf1)])) reste)
       else (CONCaux_elt_liste sub (Sec_Ab(sub1,[(b1,cf1)])) reste);;
let rec CONCaux_liste_liste sub liste1= function
    []- >[]|
    x::liste2- >eliminer_elt((CONCaux_elt_liste sub x liste1)@(CONCaux_liste_liste sub liste1 liste2
let rec decoupe_sec (S_Ab x) = function
    []- >[]|
    (l,cf)::reste- >if ((l=INF)or(l=A_ZERO))
       then ((Sec_Ab(SA_VIDE,[(l,cf)]))::(decoupe_sec (S_Ab x) reste))
       else ((Sec_Ab((S_Ab x),[(l,cf)]))::(decoupe_sec (S_Ab x) reste));;
let rec decoupe_se (S_Ab x) = function
    []- >[]|
    l::reste- >if ((l=INF)or(l=A_ZERO)) then ((Se_Ab(SA_VIDE,[l]))::(decoupe_se (S_Ab x) reste)
    else ((Se_Ab((S_Ab x),[l]))::(decoupe_se (S_Ab x) reste));;
let CONC_s sub (Sec_Ab(sub1,LC)) (Se_Ab(sub2,LL))=
    let decoupe1=decoupe_sec sub1 LC and
    decoupe2=decoupe_se sub2 LL in
    CONCaux_liste_liste sub decoupe2 decoupe1 ;;
let CONC sub (Sec_Ab(sub1,ac))(Se_Ab(sub2,a))=
    if not(NOT_EXCLUSIVE sub sub1 sub2) then [(Se_Ab(sub2,a));SEQ(Sec_Ab(sub1,ac))]
    else CONC_s sub (Sec_Ab(sub1,ac))(Se_Ab(sub2,a));;
let rec CON sub se = function
    []- >[]|
    sec::reste- >union (CONC sub sec se) (CON sub se reste);;
let rec CONC_S sub l_sec = function
    []- >[]|
    se::reste- >union (CON sub se l_sec) (CONC_S sub l_sec reste);;
let CONCSS sub =fun
```

```
    x []- >(SEQS x)|
    [] x- > x |
    x y- >CONC_S sub x y;;


let tester =fun
    (S_Ab((M m),(T t),(L (l1,l2)))) (S_Ab((M m1),(T t1),(L (l11,l12))))- >
    (m=m1)&(t=t1)|
    SA_VIDE SA_VIDE- >true|
    _ _- >false;;

let rec retire (Se_Ab(x,lc)) = function
    []- >if x=SA_VIDE then [] else [(Se_Ab(x,lc))]|
    (Se_Ab(y,c))::reste- >if (tester x y) then (Se_Ab(y,c))::(retire (Se_Ab(x,lc)) reste)
       else retire (Se_Ab(x,lc)) reste;;
let transfor l =
    let rec premier= function
    []- >failwith"il y a un probleme"|
    (Se_Ab(x,lc))::ss- >x and
    []- >[] |
    (Se_Ab(x,lc))::ss- >addition_l(lc@(secon ss)) in
[(Se_Ab(premier l,secon l))];;
let transfo l= if not(l=[]) then transfor l else [];;
    x::reste- > let p=retire x reste in
       (transfo p)@(CONCsss(subtract reste p));;
let CONCS sub sec se=CONCsss (CONCSS sub sec se);;
```

B.8 Manipulations de *sat* et de *dp*

SIGNATURES:

modified :'a * 'b ↦ (('a * 'b) * Seq_Abst list) list ↦ (('a * 'b) * Seq_Abst list) list ↦ ('a * 'b) list = < *fun* >

EXTEND : 'a * 'b ↦ (('a * 'b) * Seq_Abst list) list ↦ (('a * 'b) * Seq_Abst list) list = < *fun* >

ADJUST : 'a * 'b ↦ (('a * 'b) * Seq_Abst list) list ↦ (('a * 'b) * Seq_Abst list) list ↦ (('a * 'b) * Seq_Abst list) list = < *fun* >

trans_dp : 'a * 'b ↦ (('a * 'b) * ('a * 'b) list) list ↦ ('a * 'b) list =< *fun* >

REMOVE_DP : (('a * 'b) * ('a * 'b) list) list ↦ ('a * 'b) list ↦ (('a * 'b) * ('a * 'b) list) list = < *fun* >

EXT_DP : 'a * 'b ↦ (('a * 'b) * 'c list) list ↦ (('a * 'b) * 'c list) list = < *fun* >

ADD_DP : 'a * 'b ↦ 'c * 'd ↦ (('a * 'b) * ('c * 'd) list) list ↦ (('a * 'b) * ('c * 'd) list) list = < *fun* >

```
let egalite_se=fun (Se_Ab(x,ac)) (Se_Ab(x1,ac1))- >((x=x1)&(ac=ac1))|
    SEA_VIDE SEA_VIDE- >true|
    SEA_VIDE (Se_Ab(x,ac))- >false|
    (Se_Ab(x,ac)) SEA_VIDE- >false;;
let rec egal_se_elt x= function
    []- >false|
    y::reste- >if egalite_se x y then true else egal_se_elt x reste;;
let egalite_se_elt_inv l x=egal_se_elt x l;;
let egal_se_liste liste1 liste2=not(mem false (map (egalite_se_elt_inv liste1) liste2));;
let rec chercher_se (b,aa)= function
    []- >[]|
    ((c,d),l)::k- >if ((aa=d) & (b=c)) then l else chercher_se (b,aa) k;;
```

```
let compare (a,b) sat sat1= egal_se_liste (chercher_se (a,b) sat) (chercher_se (a,b) sat1);;
let rec enlever (a,b)= function
    []- >[](*failwith"Ce couple n'existe pas dans la liste"*)|
    ((c,d),l)::k- >if ((a=c) & (b=d)) then k
       else ((c,d),l)::(enlever (a,b) k);;

let modified (a,b) sat sat1= if (compare (a,b) sat sat1) then [] else [(a,b)];;
let EXTEND (a,b) sat=if existe_cple_c (a,b) sat then sat
    else ((a,b),[(Se_Ab(SA_VIDE,[INF]))])::sat;;

let compa (Se_Ab(sub1,LL1)) (Se_Ab(sub2,LL2))=(sub1=sub2);;
let sup_Aa=fun
    DEUX_P x- >true|
    UN_S x- > if (x=INF) or (x=UN_S) then true else false|
    UN_P x- > if (x=INF) or (x=UN_S) or (x=UN_P) then true else false|
    DEUX x- > if (x=DEUX_P) then false else true|
    INF x- > if (x=INF) then true else false;;
let ver l_new a=mem false (map (sup_Aa a) l_new);;
let ver1 l_old a=mem true (map (sup_Aa a) l_old);;
let verify l_new l_old=not (mem false (map ( ver l_new) l_old));;
let verify1 l_old l_new=not (mem false (map ( ver1 l_old) l_new));;
let ELARGISSEMENT_Aa l_new l_old =
    if (verify l_new l_old)&(verify1 l_old l_new)
    then l_new else (union l_new l_old);;
let rec group= function
    [x]- >x|
    x::y::[]- > groupe x y |
    x::y::l- >groupe (groupe x y) (group l);;



let rec compa_l y = function
    []- >[y]|
    x::reste- >if x=SEA_VIDE then compa_l y reste else
       if compa x y then x::compa_l y reste else
       compa_l y reste;;
let rec compa_l1 l= function
    []- >[]|
    x::reste- >if x=SEA_VIDE then (compa_l1 l reste)
       else (compa_l x l)::(compa_l1 l (subtract reste (compa_l x l) ));;
let compa_ll l=eliminer_elt(map group (compa_l1 l l));;

let ADJUST (a,b) sat sat1=
    if (compare (a,b) sat sat1) then sat else
```

```
        ((a,b), subtract (union (chercher_se (a,b) sat) (chercher_se (a,b) sat1))

        [(Se_Ab(SA_VIDE,[INF]))] )
let enlever_iff (a,b) dp=if existe_cple_c (a,b) dp then (enlever (a,b) dp) else dp;;
let rec trans_dp (a,b) dp =
    let cc=chercher_cple (a,b) dp in
    eliminer_cple(cc @ fouiller (enlever_iff (a,b) dp) cc) and
fouiller dp = function
    []- >[]|
    (c,d)::reste- > (trans_dp (c,d) dp )@(fouiller (enlever_iff (c,d) dp) reste);;
let remove (a,b)= function
    []- >[]|
    ((c,d),l)::k- >if existe_cple_s (c,d) (trans_dp (a,b) (((c,d),l)::k)) then k
       else ((c,d),l)::k;;
let rec REMOVE_DP dp= function
    []- >dp|
    (a,b)::reste- >REMOVE_DP (remove (a,b) dp) reste;;
let EXT_DP (a,b) dp=((a,b),[])::dp;;
let rec ADD_DP (a,b) (c,d) = function
    []- >[]|

    ((f,g),l)::k- >if ((f=a) & (g=b)) then ((a,b),(c,d)::l)::k
       else ((f,g),l)::(ADD_DP (a,b) (c,d) k);;
```

Annexe C

L'algorithme d'interprétation abstraite

SIGNATURES:

solve : Sub_Abst $\mapsto$ string $\mapsto$ Procedure list $\mapsto$ (((Modes * Types * Liaisons) * string) * Seq_Abst list) list * (((Modes * Types * Liaisons) * string) * ((Modes * Types * Liaisons) * string) list) list * int = $< fun >$

solve_call : Sub_Abst * string $\mapsto$ Procedure list $\mapsto$ ((Modes * Types * Liaisons) * string) list $\mapsto$ (((Modes * Types * Liaisons) * string) * Seq_Abst list) list $\mapsto$ (((Modes * Types * Liaisons) * string) * ((Modes * Types * Liaisons) * string) list) list $\mapsto$ int $\mapsto$ (((Modes * Types * Liaisons) * string) * Seq_Abst list) list * (((Modes * Types * Liaisons) * string) * ((Modes * Types * Liaisons) * string) list) list * int = $< fun >$

solve_clause : Sub_Abst * Clause $\mapsto$ Procedure list $\mapsto$ ((Modes * Types * Liaisons) * string) list $\mapsto$ (((Modes * Types * Liaisons) * string) * Seq_Abst list) list $\mapsto$ (((Modes * Types * Liaisons) * string) * ((Modes * Types * Liaisons) * string) list) list $\mapsto$ int $\mapsto$ Seq_Abst_Cut list * (((Modes * Types * Liaisons) * string) * Seq_Abst list) list * (((Modes * Types * Liaisons) * string) * ((Modes * Types * Liaisons) * string) list) list * int = $< fun >$

solve_procedure : Sub_Abst * Clause list $\mapsto$ Procedure list $\mapsto$ ((Modes * Types * Liaisons) * string) list $\mapsto$ (((Modes * Types * Liaisons) * string) * Seq_Abst list) list $\mapsto$ (((Modes * Types * Liaisons) * string) * ((Modes * Types * Liaisons) * string) list) list $\mapsto$ int $\mapsto$ Seq_Abst list * (((Modes * Types * Liaisons) * string) * Seq_Abst list) list * (((Modes * Types * Liaisons) * string) * ((Modes * Types * Liaisons) * string) list) list * int = $< fun >$

```
let rec solve (S_Ab s_in) nom_proc program =
    solve_call ((S_Ab s_in),nom_proc) program [] [] [] 0 and
solve_call ((S_Ab s_in),nom_proc) program suspended sat dp iter =
    if ((existe_cple_c (s_in,nom_proc) dp)or
    (existe_cple_s (s_in,nom_proc) suspended))
    then (sat,dp,iter)
    else
    let iter1=iter+1 in
    let sat_int= if (not(existe_cple_c (s_in,nom_proc) sat)) then
       (EXTEND (s_in,nom_proc) sat)
       else sat in
    let rec boucle ((S_Ab s_in),nom_proc) program suspended sat_int dp iter=
    let dp_int= eliminer_cple(EXT_DP (s_in,nom_proc) dp) in
    let suspended_int=eliminer_cple(suspended@[(s_in,nom_proc)]) in
    let (B,sat_int1,dp_int1,iter_int1)=
    solve_procedure ((S_Ab s_in),rech_pro nom_proc program) program
    suspended_int sat_int dp_int iter in
    let sat_int2=ADJUST (s_in,nom_proc) sat_int1 [((( s_in),nom_proc), B)] in
    let modify=modified (s_in,nom_proc) sat_int1 [((( s_in),nom_proc), B)] in
    let dp_int2=REMOVE_DP dp_int1 modify in
    if (existe_cple_c (s_in,nom_proc) dp_int2) then (sat_int2,dp_int2,iter_int1) else
    boucle ((S_Ab s_in),nom_proc) program suspended_int sat_int2 dp_int2 (iter_int1+1) in
    boucle ((S_Ab s_in),nom_proc) program suspended sat_int dp iter1 and
solve_clause ((S_Ab s_in),claus) program suspended sat dp iter =
    let C=EXTC claus (S_Ab s_in) in
    let rec exec_corps (Sec_Ab se_in) sat dp iter=function
    []- >([(Sec_Ab se_in)],sat,dp,iter)|
       at::reste- >let n =type_atome at in
           if (n=9) then exec_corps_l sat dp reste iter [(AI_CUT (Sec_Ab se_in))] else
           let s_int=RESTRG at (SUBST (Sec_Ab se_in)) in
           if (n=8) then exec_corps_l sat dp reste iter [(rendre_sec(UNIF_VAR at s_int))] else
           if ((2¡=n)&(n¡=7)) then
           exec_corps_l sat dp reste iter [(rendre_sec(UNIF_FUNC at s_int))] else
           let nom_proc=name_proc_appel at in
           let (sat_int,dp_int,iter_int)=solve_call ( s_int,nom_proc) program suspended sat dp iter in
           let nom_clause=name_clause claus and s_int1=triplet s_int in
           let B=chercher_se (s_int1,nom_proc) sat in
           let dp_int1=if existe_cple_c (s_int1,nom_proc) (dp_int)
           then ADD_DP (s_in,nom_clause) (s_int1,nom_proc) dp_int
           else dp_int in
           exec_corps_l sat_int dp_int1 reste iter_int (EXTGS at (Sec_Ab se_in) B) and
           exec_corps_l sat dp l_at iter=
```

```
        let union_3 (a,b,c,iter) (f,d,e,iter1)=(union a f, union b d, union c e,iter1) in
        function        []- >([],[],[],iter)|
          sec::k- >union_3 (exec_corps sec sat dp iter l_at)
          (exec_corps_l sat dp l_at iter k) in
        let (p1,p2,p3,iter_int)=(exec_corps C sat dp iter (corp ( claus))) in
        (RESTRCS claus p1, p2, p3,iter_int) and
solve_procedure ((S_Ab s_in),proc) program suspended sat dp iter=
  let claus=hd proc and reste=tl proc in
  let (C,sat_int,dp_int,iter_int)=solve_clause ((S_Ab s_in),claus) program suspended sat dp iter in
  let B=SEQS(C) in
  let ncu=(cut_or_nts C) in
  if ((reste=[]) or (ncu= true)) then (B, sat_int,dp_int,iter_int) else
  let (B_int,sat_int1,dp_int1,iter_int1)=
  solve_procedure ((S_Ab s_in),reste) program suspended sat_int dp_int iter_int in
  (CONCS (S_Ab s_in) C B_int,sat_int1,dp_int1,iter_int1);;
```

Annexe D

Exemple testé

```
#open"sys";;
   let test =
(* La procedure nat*)
        let nat=(PR(("nat",[(CL((TE("nat",[1])),(CO[(FU ((VAR 1),ZERO))])));
           (CL((TE("nat",[1])),(CO[(FU ((VAR 1),s(VAR 2)));
           (AP("nat",[2]))]))))])))) in
(* La procedure nat*)

        let nat1=(PR(("nat1",[(CL((TE("nat1",[1])),(CO[(FU ((VAR 1),ZERO))])));
           (CL((TE("nat1",[1])),(CO[(FU ((VAR 1),s(ZERO)))])));
           (CL((TE("nat1",[1])),(CO[(FU ((VAR 1),s(VAR 2)));
           (AP("nat1",[2]))]))))])))) in
(* La procedure natPlus*)
        let natPlus=(PR(("natPlus",[(CL((TE("natPlus",[1;2;3])),
           (CO[(AP("nat",[3]));(AP("natPlusAux",[1;2;3]))])))])))) in
(* La procedure natPlusAux *)
        let natPlusAux=(PR(("natPlusAux",[(CL((TE("natPlusAux",[1;2;3])),
           (CO[(FU ((VAR 1),ZERO));(UN ((VAR 2),(VAR 3)))])))]));
           (CL((TE("natPlusAux",[1;2;3])),
           (CO[(FU ((VAR 1),s(VAR 4)));(FU ((VAR 2),s(VAR 5)));
           (AP("natPlusAux",[4;2;5]))])))])))) in
(* La procedure natMul**)
        let natMul=(PR(("natMul",[(CL((TE("natMul",[1;2;3])),
           (CO[ (AP("natPlus",[1;2;4])); (AP("natMulgga",[1;2;3]))])))])))) in (* La procedure natMul**
        let natMulgga=(PR(("natMulgga",[(CL((TE("natMulgga",[1;2;3])),
           (CO[(FU ((VAR 1),ZERO));(FU ((VAR 3),ZERO));CUT])));
           (CL((TE("natMulgga",[1;2;3])),(CO[(FU ((VAR 1),s(VAR 4)));
           (AP("natMulgga",[4;2;5]));
```

```
        (AP("natPlusAux",[2;5;3]))]))]))) in
(*Le programme a tester*)
      let pgm=([nat;nat1;natPlus;natPlusAux;natMul;natMulgga]) in
(*Les substitutions abstraites d'entree*)
   let B_in=(S_Ab((M[(1,AA)]),(T[(1,QQ)]),
      (L([],[(1,[])]))))  in

   let B1_in=(S_Ab((M[(1,AA);(2,AA);(3,AA)]),(T[(1,QQ);(2,QQ);(3,QQ)]),
      (L([],[(1,[]);(2,[]);(3,[])])))) in

   let uni (a,l)=list_length l in
   let unir k=map uni k in
   let rec regrouper =function
      []- >0|
      a::k- >a +regrouper k in

   let lettemps=solvegrsubproc(unigmpi) inlet temps1 = time() in
      let rec aff n=function
      []- >[]|
      ((a,b),x)::reste- >(n,"ENTREE=",(a,b),"RESULTAT=",x)::
      aff (n+1) reste in
      let rec test i =

      let (sat,dp,iter) = solve sub proc pgm in
      if (i-1)=0 then (time(),sat,dp,iter) else
      test (i-1) in
      let (temps2,sat1,dp1,iter1) = test i in
      let temps=temps2 -. temps1 in
      let n=list_length sat1 in
   (i,dp1,"EXECUTIONS DE",proc," ONT PRIS AU TOTAL ",temps,

   "SECONDES","POUR",iter1,"ITERATIONS.

   VOICI LES RESULTATS:", (aff 1 (arr(arranger(fusion3 sat1))))) in
   test_solve B1_in "natMul" pgm 60;;
```